Cleared version of October 8, 1979

ALTO SUBSYSTEMS

Compiled on: October 8, 1979

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Alto Subsystems

This document is a directory of major Alto BCPL subsystems. Mesa subsystems are collected together and documented elsewhere.

Binary versions of these programs are available on the <Alto> directory. If the documentation for the subsystem is short, it is included in this file directly. If it is somewhat longer, the documentation is stored separately and the entry is marked with a *. The documentation for these objects is available on <AltoDocs> in .TTY files. Programs that have quite bulky documentation are denoted by **. These programs have separate documentation on <AltoDocs>, usually as <AltoDocs>Name.press.

If you would like a full listing of documentation for all but the ** programs give the command "Press <AltoDocs>Subsystems.press".

The person last known to be responsible for each subsystem is also given.

*ASM: an assembler for Alto machine language, producing object files compatible with the Bcpl loader. (Ed McCreight)

**BCPL: a compiler for the Bcpl language. (Dan Swinehart)

**BLDR: a loader for object files produced by Bcpl and Asm. It is documented in the Bcpl manual. (Dan Swinehart)

**BRAVO: a display editor. (Charles Simonyi)

*BUILDBOOT: a program for constructing Alto boot files. (David Boggs)

*CHAT: establishes PUP Telnet connections between a pair of cooperating parties. (Ed Taft)

CLEANDIR: does a garbage collection on a file directory (not on the disk space, though). Call it with
>CLEANDIR directory-name n
to clean up the specified directory. The system directory is called SYSDIR. The second parameter, n, tells how much extra space to append to the directory. The reason for it is that extending the directory in this way will tend to get the pages allocated to consecutive disk sectors, so that subsequent lookups will go faster. If this program fails, it will leave your disk unusable. To guard against this, you can copy SYSDIR to a dummy file and run CLEANDIR on that first, then run it again on SYSDIR. DO NOT try to copy the cleaned-up dummy file back to SYSDIR. (David Boggs)

*COPYDISK: copies whole Diablo and Trident disk packs from one drive to another on the same Alto or through the net between two Altos. (David Boggs)

*CREATEFILE: creates a file of a given size, trying to allocate it contiguously. (David Boggs)

*DDS: The Descriptive Directory System is a front end for the Alto file system, providing a relational data base management system and facilities for displaying information related to Alto files. (Peter Deutsch)

*DMT/PEEK/PEEKSUM: Alto memory diagnostic program and related statistics-gathering programs. (David Boggs)

*DPRINT: Prints disk files on the Diablo Printer. (Ed Taft)

**DRAW: An illustrator. (Patrick Beaudelaire)

EMPRESS: Converts ordinary text files to Press files, and performs simple formatting operations, intended for listing programs. (David Boggs)

*EXECUTIVE: The Alto command processor. (Richard Johnsson)

*FIND: a program to search text files for user-supplied strings. This program originated as a demonstration of the power of compiling microcode from the given problem. (Peter Deutsch)

*FTP: a Pup-based File Transfer Program for moving files to and from an Alto file system. (David Boggs)

*LISTSYMS: converts the .Syms file produced by BLDR into human readable form. (Peter Deutsch)

*MAILCHECK: A program that will check for waiting mail on Maxc. (Larry Masinter)

**MARKUP: A document illustrator. (William Newman)

MICRO: The microcode assembler for Maxc, Dorado, D0, and other machines. Basic documentation is available only in the CSL archives. It is called "Maxc document 9.2". Recent changes are documented in <AltoDocs>Micro.tty. (Peter Deutsch)

MOVETOKEYS: Moves page 1 of the named file to the appropriate page of the disk so that depressing the key-combination and the boot button will boot-load the file. (Roy Levin)

*MU: The microcode assembler for the Alto. (Ed Taft)

Neptune: A program for listing, copying, and deleting files. It is capable of dealing with both drives of a two-drive Alto. The program offers help on its use. Documentation is in the Alto User's Handbook. (Keith Knox -- WRC)

*NETEXEC: This subsystem, which is bootstrapped over the Ethernet, provides a convenient interface to the other systems available from "boot servers" on the network. (David Boggs)

NEWOS.BOOT: is the name of a ready-to-install Operating System. Retrieve it, say "Install NewOS.boot" to the Exec, and then delete it (it writes itself out on the file Sys.boot) (David Boggs)

*OEDIT: allows you to look at and modify arbitrary files in octal. (Dan Swinehart)

*ORAM: A scheme for overlaying several segments of microcode in the Alto RAM. (Peter Deutsch)

*PACKMU/RPRAM: These two subsystems, in conjunction with the subroutine ReadPRAM or LoadRam, allow programs using the RAM to check the constant memory and load the RAM as a part of their initialization. (Peter Deutsch)

*PEEKPUP: a Pup software debugging aid. (David Boggs)

**PREPRESS: A program for manipulating font files. (Joe Maleson)

*PRESSEDIT: combines Press files, converts Ears files into Press format, or adds extra fonts to a Press file. (William Newman)

PROOFREADER: Proofreader for English text. (Ed McCreight)

*RAMLOAD: a program for loading the Alto RAM from the files produced by the microcode assembler, MU. (Dave Boggs)

READPRESS: reads Press files and displays a text-listing of the entity commands, DL strings, etc. Command line is of the form: "ReadPress Test.Press". (Joe Maleson)

*SCAVENGER: a subsystem for repairing a damaged Alto file system. (Richard Johnsson)

**SIL, Analyze, Route, Build, NetDelays, etc.: A system for automating logic design, including an illustrator specialized to logic drawings. (Roger Bates, Ed McCreight)

SORT: a very small subsystem which will sort files containing less than 1000 entries, each terminated by a carriage return. Call it with

>SORT <sortfilein> <sortfileout>

If <sortfileout> is omitted, the sorted data will be written back to <sortfilein>.

*SWAT: a debugger for Bcpl programs. (David Boggs)

SYS.BOOT: is the name of the boot file for the operating system on the Alto disk. (David Boggs)

*Trident disk software: TFU, TRIEX and the TFS software package. The Bcpl software package and utility programs for driving Trident disks interfaced to the Alto. (Ed Taft)

*VIEWDATA: a subsystem that displays 2D projections of 3D data on the Alto screen. (Dick Lyon)

MISCELLANEOUS PROCEDURES AND INFORMATION
FOR PARC ALTO USERS

*NEWDISK: a procedure for creating a virgin disk and getting fresh, up-to-date software from MAXC. (David Boggs)

*PARCALTOS: a document containing miscellaneous information for Alto users and maintainers at PARC.

ASM

This assembler, written in BCPL, runs on the Alto and produces BCPL-compatible relocatable binary output files, suitable for input to BLDR, the BCPL loader. The Alto Hardware manual describes the source language and the virtual machine.

1. Symbols

Symbols may be up to 130 characters in length, and every character of a symbol must be used to identify it. By default upper- and lower-case characters are different, and two character strings represent the same symbol only if the same letters and cases are used in both. However, the /U switch causes all lower-case letters in symbols to be changed to upper case (even in external symbols). Thus if you want an assembly-language program to link to symbols containing lower-case letters, you must either default lower-case conversion in ASM or map all symbols to upper case in BLDR using its /U switch.

2. Strings

Strings follow BCPL conventions. They may not extend from one line to the next.

3. Assembly Regions

This assembler can assemble into three regions: two static regions (one in page 0) and one code region. The directives .NREL, .SREL, and .ZREL cause the assembler to begin placing code in the code region, the non-page-0 static region, and the page 0 static region, respectively. The BCPL loader causes the restrictions that the code area may not contain pointers into the code area, that the first word of the code area may not point to a static area, and that no static area may contain pointers to a static area. The only external symbols are statics.

Arithmetic is not allowed on symbols denoting statics, and the symbol "." is undefined in .SREL and .ZREL. Any absolute or code- relative expression (including such goodies as JMP@ 62) may be placed in .SREL or .ZREL. Any absolute expression, static reference, or instruction reference to .ZREL may appear in .NREL.

4. Text

There are two text modes, .TXTM B and .TXTM L. Mode B causes the generation of standard BCPL strings. Mode L causes the generation of long strings, a full word length followed by the string characters, two per word.

5. .GET

The directive .GET "FOO" causes the file FOO to be inserted into the source text at that point. .GET can be used up to two levels deep. Its primary utility is likely to be for lists of externals and for canned entry and exit sequences.

6. .GETNOLIST

Works exactly like .GET, except that the "gotten" file is not included in the listing, nor are any files which it .GET's.

7. .BEXT

In addition to .EXTN and .EXTD and .ENT, I have added two directives .BEXT and .BEXTZ which work exactly as BCPL's External works for non-page-0 and page 0 statics, respectively. This should increase the utility of the .GET feature above.

8. Expressions

Parentheses (but not precedence) are supported. Constructs like "K and $*N and 5 and 17. and 3B10 are all primaries. Most BCPL and customary assembler operators are allowed. The construct 1B10 means 40(octal), unlike BCPL's convention. I am willing to be convinced on this point.

9. Predefined Symbols

All predefined symbols and directives and opcodes are defined both in all upper-case and all lower-case letters. For example, both LDA and lda are predefined, but Lda is not. The following Alto-specific opcodes are preloaded in the symbol table:

```
JSRII  JSRIS  CYCLE  CONVERT DIR   EIR   BRI
RCLK   SIO    BLT    BLKS    SIT   RDRM  WTRM
JMPRM  MUL    DIV
```

In addition, the following pile of skips which test various conditions has been added, courtesy of Dan Ingalls. Only the names have been changed to confuse the innocent:

Two operands:

```
SZE   SZ    SNZ   SP    SGZ   SN    SEQ
SE    SNE   SLT   SLE   SGT   SGE   SGTU
SLEU  SGEU  SLTU  SODD  SKEVEN SNIL  SNNIL
MKZERO MKONE  MKNIL  MKMINUSONE
```

No Operands:

```
NOP   SKIP
```

It should be explained that U stands for unsigned, and that Dan thinks of NIL as -1.

10. Operation

If the source file is called FOO.ASM, type

ASM FOO.ASM

If you just type ASM FOO it will first try to use FOO and, failing in that, try FOO.ASM. The assembler will usually want to construct several files, which it will do by substituting various extensions on FOO unless you specify otherwise. There are a lot of switches which apply to ASM:

/L Construct a listing file
/S Include the symbols defined by the user, for what they're worth
/A Include all symbols, even the predefined ones
/R Include a printout of the .BR file
/N Don't make a .BR file
/E Make an .ER file which is a copy of the error messages sent to the terminal
/D Print debugging messages (as errors, in fact)
/P Pause after printing each error message (continue with CR)
/U Map all lower-case letters in symbols to upper-case

There are also a lot of switches which apply to file names, and which tell the assembler to use this name instead of the one it was about to invent:

/L Names the listing file
/E Names the error file
/S Names the source file (also no switches)
/T Names the temporary file
/B Names the relocatable binary file

Alto Boot Files: Formats and Construction

The process of "booting" the Alto is one of setting some or all of the Alto's state either by reading a file from the disk or by accepting packets from the Ethernet. This document attempts to explain the various ways that state is restored, and the formats of "boot files" built by various programs.

There are four basic steps in "booting" the Alto: (1) the tasks in the microprocessor are reset; (2) a 256-word "boot loader" is loaded into main memory and started; (3) the boot loader loads a portion of Alto main memory from a "boot file" and finishes by transfering to a known place; (4) the user's program loaded by the third step can restore even more of the Alto's state.

1. Booting

"Booting" is accomplished either by pushing the "boot button" located on the rear of the keyboard or by executing the SIO instruction (see Alto Hardware Manual). Unless overridden by the Reset Mode Register, the emulator task is started in a standard boot program. This program reads location 177034b, a word whose contents can be altered by pushing various keys on the keyboard. If the <bs> key is depressed during booting, the machine state will be restored from the Ethernet; otherwise, the state is restored from the disk.

When booting from the disk, the keyboard word is interpreted as a disk address where a "disk boot loader" is located. If no keys are depressed, disk address 0 is generated, which is the normal resting place of the "disk boot loader" for the operating system. The emulator reads a single 256-word disk record into memory locations 1, 2, ...400b; the 8-word disk label for this page is placed in 402b, 403b, ... 411b. When the disk transfer is complete, control is transferred to location 1 in the loader. The boot loader uses the saved label to point to the remainder of a "boot file" which is read into main memory and started. The types of "disk boot loaders" and "boot files" are discussed below.

When booting from the Ethernet, the microcode waits until a "breath of life" packet arrives, containing a 256-word "Ethernet boot loader" which is read into locations 1 - 400b and executed by transferring to location 3. It is up to this loader to establish communications with a party willing to deliver the remainder of the state needed.

2. Boot File Formats and Boot Loaders

There are two basic kinds of boot files, and a variant:

B-File: Built by the BuildBoot program; loader is DiskBoot.

S-File: Built by the OutLd subroutine; "S" loader.

S0-File: Variant of S-File built by the SaveState subroutine.

A B-File can be distinguished from an S-File or S0-File because B-Files have a 0 in their second data word.

Words 4 & 5 of B, S, and S0 boot files do not contain code and are reserved for holding the (Alto format) date on which the file was built. Boot servers use this information to propagate the latest versions. Old format type B files which don't contain a date have 402b in file word 0. Old format type S files have 355b in file word 0.

2.1. B-Files

B-Files ("BuildBoot" files) are the simplest sort of boot file. The booting process itself does not restore the entire state of the machine; page 1 (addresses 400b to 777b) is not restored; no RAM or R-register state is restored except for the program counter.

A boot loader resides in the first (256-word) data page of a B-File. It is this page that is read in by the booting process. The file is formatted as follows:

```
File page 1   =>   DiskBoot loader
File page 2   =>   Image of memory page 0 (0-377b)
File page 3   =>   Image of memory page 2 (1000b-1377b)
File page 4   =>   Image of memory page 3 (1400b-1777b)
   ...                ...
File page n   =>   Image of memory page n-1
```

The file can be of any length, except that n must not exceed 254. After reading the entire file, control is transferred to the restored state by doing JMP@ 0.

2.2. S-Files

S-Files ("Swat" files) are a somewhat complicated construction that permits more of the Alto state to be restored: the interrupt system, active display, and so forth are all restored. In order to achieve this, the restored state must contain a copy of the OutLd subroutine that is responsible for the final stage of the restore; when the state is fully restored, this subroutine simply returns to its caller. This full state save and restore was originally designed for the Swat debugger. (Note: no RAM or R-register state except for the PC and accumulators is restored by this kind of boot.)

A boot loader resides in the first (256-word) data page of an S-File. This is the page read by the booting process. The file looks like:

```
File page 1     =>   "S" loader
File page 2     =>   Image of memory page 2 (1000b-1377b)
File page 3     =>   Image of memory page 3 (1400b-1777b)
   ...                  ...
File page 253   =>   Image of memory page 253 (176400b-176777b)
File page 254   =>   Image of memory page 1 (400b-777b)
File page 255   =>   Image of memory page 0 (0-377b)
```

The S-File must contain at least 255 data pages; additional pages are ignored by the booting process, and can be used to save additional state. When the restore is finished, control returns to the caller of OutLd (see Alto Operating System Manual).

ssec(S0-Files)

S0-Files are a minor variant of S-Files that can be used to restore the Alto state in 2 different ways. The variation is simply that location 0 of the restored memory image (i.e., word 0 of file data page 255) contains an "alternate starting address." The file can be loaded by (1) using it as an S-File, and executing the loader saved in its first file data page, or (2) by a loading process that loads all memory but page 1 (file page 254) and does a JMP@ 0. The operating system boot file, Sys.Boot, is an S0-File.

The S0-File is designed to permit Ethernet booting from states conveniently saved by OutLd.

2.3. DiskBoot loader: B-Files

The DiskBoot loader is commonly placed as the first data page in B-Files. Its source is DiskBoot.Asm (in BuildBoot.Dm); BuildBoot will normally include this loader on the front of the B-Files it constructes. NOTE: the file "DiskBoot.Run" is not a literal copy of the 256 words that go on the front of the file, but the result of applying Bldr to the relocatable file generated by assembling DiskBoot.Asm. B-files were the first boot format designed for the Alto. Unlike an S-file which must be at least 255 data pages long, a B-file need be big enough to contain all of the code to be loaded.

2.4. InOutLd loader: S-Files and S0-Files

This loader is part of the Operating System and available as a separate package. For more details read the descriptions of InLd, OutLd and BootFrom in the Alto Operating System manual.

2.5. EtherBoot loader: "Breath Of Life"

The "breath of life" loader, which is periodically broadcast by gateways, is loaded into locations 1-400b when the Alto is booted with the <bs> key pressed. The standard form of this loader reads location 177035b (a keyboard word), and transmits "MayDay" packets containing the 16-bit result. Some server on the network will interpret the 16-bit argument as a request for a specific program. The server will open an EFTP connection with the Alto which sent the MayDay. It transmits data pages in the same order as they are recorded in B-Files (including the first data page, even though it contains a disk-oriented loader). When the connection is closed, the loader starts the restored image by doing a JMP@ 0.

By convention, the 16-bit argument 177777b is never answered by a server. This convention is used by programs which have specifically started a "breath of life" loader and are expecting an EFTP connection from some specific party.

The EtherBoot loader is available as a package: see the Alto Packages manual. Protocol details are in the Pup documentation.

3. Constructing B-Files: BuildBoot

BuildBoot.Run constructs files for direct booting into the Alto. The program copies its input files into an output file according to directives in the command line and in the input files themselves. Two kinds of input files are supported at the moment. One is the segment file, which contains a block of words to be loaded into contiguous addresses. The other is the executable (.Run) file, which is what Bldr produces on the Alto (see Alto Operating System Reference Manual for details). If several files in the command line specify the contents of the same memory location, the last one will win. In addition to the data already in the output file, the program maintains four state variables between items in the command line. One is the location counter which specifies the address where the next segment file (if any) will be placed. Another is the address where the loaded image is to begin execution. This defaults to the starting address of the last executable file in the command line. The third is the address (if any) where the layout vector of the next executable file is to be loaded. If this address is missing, the layout vector will not be loaded. The fourth is the address (if any) in the boot loader where the current date and time will be placed.

Here are the switches:

/E	This is an executable file (also no switches or /R)
/D	This is the address of a two word block in the boot loader where the current date and time are placed.
/S	This is a segment file
/N	Reset the location counter to this octal number
/O	This is the output file
/G	This octal number specifies where execution begins

/B This executable file contains a boot loader in its code area. If omitted, defaults to "DiskBoot.Run"
/L Write load map on this file
/V The layout vector of the next executable file will be loaded in a contiguous block starting at the address specified by this octal number

If we wanted to bootify the .Run file Prom.run, we might say

```
BuildBoot Prom.boot/O Prom.map/L 20/N 1000/G↑
    Prom.run/S
```

Similarly, if we had the diagnostic DMT.RUN as an executable file (including any runtime support it might need), we could simply say

```
BuildBoot DMT.boot/O DMT.DMT.map/L DMT.run/E
```

The disk boot loader DiskBoot.Run is also included in the file BuildBoot.Dm, and is required by BuildBoot unless another boot loader file is specified by the /B switch.

The BootBase package (<AltoSource>BootBase.dm) makes it possible to construct a B-format boot file out of most any .Run file without any souce-level changes. It initializes an execution environment; provides a runtime environment including TeleSwat, the Bcpl runtime routines, Calendar clock maintenance, parity error handling; and supplies selected Operating System routines.

Two standard configurations are available: BasicBoot is a bare bones Bcpl environment suitable for diagnostics; FullBoot adds most of the facilities of the Alto Operating System except for the BFS, Disk Streams, and Directories. Other configurations are straight forward. Each configuration consists of four files: xBootBase.run (x = Basic or Full) contains code. xBootBase.bj contains Bldr linkage information similar to Sys.bk. xBootBase.xc contains part of the Bcpl runtime. LoadxBoot.cm is a command file template containing incantations to Bldr and BuildBoot and slots which you must fill in.

4. Constructing S-Files: OutLd

S-Files are constructed by the OutLd subroutine, which is documented in the Alto Operating System Manual.

5. Constructing S0-Files: SaveState

The SaveState subroutine, also included in BuildBoot.Dm, can be called in a fashion similar to OutLd, but it will create an S0-File. The Bcpl call is:

```
SaveState(filename, [flags])
```

It behaves like OutLd in that it returns 0 if the file has just been written, 1 if it has been restored by an InLd, 2 if by a disk boot, and (unlike OutLd) 3 if by an EtherBoot. If bit 15 of flags is set, the disk state is flushed after creating the boot file. If bit 14 is set, the disk state is recomputed when the boot file is started. SaveState requires the presence of operating system levels through disk streams.

6. The "standard boot file": disk address 0

The 256-word data page saved on real disk address 0 cannot be part of any legal Alto file because of the way the file system is designed. As a result, the standard boot file is established by copying the first data page of the boot file (e.g., Sys.Boot) into disk address 0 (the label and data portions are both copied verbatim). Thus the proper data (disk boot loader) will be read when booting, and the label will point forward to the (legal) boot file, data page 2. This makes Sys.boot have an illegal format (the forward links of two pages point at page 2 of Sys.boot), but the Scavenger knows this and ignores it.

CHAT

Chat is a program for establishing Pup Telnet connections between a pair of cooperating parties. Its chief function is to permit Alto users to login to Maxc and IFS servers. Chat includes an extension to support text-display control and graphics.

1. Simple operation

Chat is organized so that default operation with Maxc1 is simple. Simply saying "Chat" will establish a connection with Maxc and (provided you are "logged in" on your Alto) will try to establish the Alto as controlling terminal for a Maxc job that is logged in under your name. Chat will perform a "login" or "attach" as appropriate. If the simple methods fail you must deal with Maxc yourself (life is hard).

To connect to some server besides Maxc, type "Chat name" where "name" is the name of the desired server (Maxc2, Ivy, DLS, etc.) Chat will perform the automatic login if the server is a Maxc or an IFS.

If you don't have the file Chat.Run on your disk, the Alto Executive will boot-load it from a boot server on the network. In this case, Chat will not use the "name" you supply on the command line but rather will require you to type the server name directly to Chat.

If you are not logged in on your Alto at the time you start Chat, or you booted Chat from the network, Chat will first request that you type in your user name (if different from the one installed on your disk) and password.

The preferred method for exiting Chat is to depress the key immediately to the right of the "return" key on the keyboard, and then to press "q" for Quit. The other method, <shift>SWAT, is frowned upon and is not guaranteed to work.

If the connection fails or is broken by the server, Chat will display an appropriate message and will ordinarily terminate. However, if you booted Chat from the network, Chat will continue running and will ask you for the name of a new server to connect to.

2. Command Interpreter

While Chat is running, you may wish to give various commands that alter its operation. Depressing the key immediately to the right of the RETURN key will cause Chat to enter a command mode. The commands are:

Q Quit--terminate the connection.

F Specify a new font. The screen will be re-initialized, which will cause recent typeout to disappear. If insufficient core space is available for the font, the system font will be used.

D Specify a "do" file to insert now. The text of the file will be treated as if it had been typed in at the keyboard--it will be transmitted to the connected party. This is a simple way to "can" Maxc procedures that you use a lot.

E Change local echo setting. Chat starts out assuming that the connected party will echo all characters. In some instances, Chat will want to echo your typein locally (e.g., when connected to another Chat).

C Change control character output setting. Control characters other than CR, LF, and Tab are normally displayed as "↑x". Changing this setting causes control characters to be thrown away.

I Toggle the "input" switch for the typescript file, set by the USER.CM entry TYPESCRIPTCHARS (see below).

O Toggle the "output" switch for the typescript file, set by the USER.CM entry TYPESCRIPTCHARS (see below).

N Permits you to establish a New connection (after breaking the current one), without leaving Chat.

3. Command-line options

Several options may be passed to Chat by global switches in the command line (i.e., by typing Chat/s/t where "s" and "t" are the switches):

/A "Attach" -- meaningful only when connecting to Maxc. This will force the Maxc attach sequence to be typed rather than whatever Chat considers appropriate.

/L "Login" -- meaningful only when connecting to Maxc or an IFS. This forces a login sequence to be typed, regardless of what Chat considers appropriate. For example, if you already have a detached job on Maxc and wish to create a new job, you must use this option.

/N Chat will not attempt any automatic login or attach.

/S Chat will be a "Pup Telnet Server," and will respond to requests for connection from others rather than initiate requests itself.

/E Chat will cause local echoing of input characters.

/C Chat will suppress output of control characters, rather than displaying them as "↑x".

/I Equivalent to the command-line entry Chat.Initial/D (see below).

/P or /D Chat will enable a display protocol (see below).

Several options may be specified with "local" switches:

string This gives the "name" of the party with whom Chat should initiate a connection. The name may be an address constant of the form net#host#socket, or may be a full symbolic name like Maxc+Telnet (see "Naming and Addressing Conventions for Pup" for details). The default socket is 1, the Telnet socket. Thus typing "Chat Regis" will try to connect to a Telnet server on the host named Regis.

filename/F Specifies the name of the font to use.

filename/D This gives a "do" file name that is fed to the connected party. When the last character of the file has been sent, Chat will not close the connection.

filename/E Similar to /D, but will end the connection when end of file is encountered.

4. USER.CM Options

The USER.CM file may also contain defaults that Chat examines at initialization. The section of USER.CM that Chat examines must begin with a line with the 6 characters [CHAT] on it. Thereafter, lines begin with "labels," followed immediately by colons, followed by arguments.

Note that Chat does not look at User.cm (or anything else on your disk) if you boot-loaded it from the network.

In the following descriptions, square brackets enclose parameters that are optional--you shouldn't actually type the square brackets.

FONT: AltoFontName.AL [width height]

Gives the name of a font to use when displaying typeout from the connected party (default: system font). If two numbers follow the name, they are interpreted as the width of a line (in characters) and the height of a page (in lines). These numbers override the calculations made by Chat, and are shipped to the server to set the terminal parameters.

BORDER: BLACK|WHITE

Gives the color of the top border of the screen (default: white).

BELL: [DING] [FLASH] [AUDIO]

Tells what to do when a bell character is received. If DING is specified, a pattern that spells out DING will be displayed at the top of the screen. If FLASH is specified, the bottom area of the screen will flash black. If AUDIO is specified, and you have a loudspeaker connected to your Alto's Diablo printer interface, an audible tone will sound. Any combination of options can be specified together (default: DING FLASH).

CONNECT: net# host# socket or host-name

Gives the network address constant or name of the party with whom a connection should be initiated (see "Naming and Addressing Conventions for Pup" for details). Default is Maxc+Telnet, the Maxc Pup Telnet server.

TYPESCRIPT: filename [length]

Gives the name of a file on which to record a typescript of the session. The file will be treated as a "ring" buffer of specified length (in bytes; default 5120). The file will be created at the beginning of the session, so that the user can be certain the disk will not overflow when recording typescript information. The string <=> will mark the end of the ring buffer, which will be updated periodically.

TYPESCRIPTCHARS: [ON|OFF] [ON|OFF]

This entry governs the selection of characters that are included in the typescript file. The first on/off switch controls characters typed on the Alto keyboard: if the switch is "on," these characters will be entered in the typescript file. The second switch controls characters sent from the other party to the Alto: if the switch is "on," these characters will be entered in the file. Default is OFF ON.

LINEFEEDS: ON|OFF

Normally, line feeds transmitted by the other party are included in the typescript file. If you wish to keep line feeds out of the file, set LINEFEEDS: OFF.

ECHO: ON|OFF

This option turns on local echoing. This is usually necessary only if you are connecting to another Alto running Chat that has used the /S option.

CONTROLCHARS: ON|OFF

Normally, control characters other than CR, LF, and Tab are displayed in the form "↑x". This option forces them not to be displayed at all. Default is ON.

DISPLAYPROTOCOL: ON|OFF

This entry enables a display protocol. The same effect can be achieved with the /P or /D command-line switches. Default is OFF.

5. Display Protocol

Chat allows a remote program to control carefully the entire Alto display. The interactive facilities of the Alto can thus be used by MAXC programs and others. A set of Interlisp-10 functions has been written to ease use of the display from LISP. These functions are documented in "Raster Graphics for Interactive Programming Environments," by R.F. Sproull, CSL-79-6, and are contained in <SPROULL>ADIS.COM; the symbolics (should you need them) in <SPROULL>ADIS.

"Display Chat" is almost completely different from "teletype Chat"; they are loaded as one program largely for convenience. To exit display Chat, use the <shift><Swat> convention. Be very careful when attaching and detaching jobs that have Chat display connections open. If you re-attach to a LISP job that previously had connections open, and CONTINUE your LISP job, the connections are no longer usable because the Pup executive has timed them out. ADISCheck should be called to verify the state of the connection. After this call, it may be necessary to invoke ADISInit again. If this procedure is not followed, you may get traps with "IO Data Error" or some such message coming out of your LISP program!

Fonts are declared in User.Cm as follows: a line of the form "DISPLAY-FONT: FileName" is a font declaration. Numbers are associated with the fonts by the order in the file: the first is font 0, the second font 1, etc. The fonts must be in "strike" format; several fonts in this format are saved on the <ALTOFONTS> directory with extension .STRIKE.

The number of "regions" available to Chat can be altered by including a line of the form "DISPLAY-REGIONS: 6" in User.Cm.

Two functions for making hard copies are not documented in the CSL report:

ADISPress[file] (Flush). This function writes a one-page Press file of the given name on your Alto disk. The page contains a bit-map for the current contents of the Chat display area. WARNING: This function requires considerable quantities of disk space (about 130 pages per file), and may lead to errors while writing the file. Best use it only when your state is safe.

ADISPressMaxc[file;scaleFactor] (Flush). This function is similar to ADISPress, but the file will be written on the connected MAXC directory. The scaleFactor defaults to 1.0, but can be set to any fraction. It will cause the Press file to contain directives to reduce the size of the image of the screen when it is printed.

Efficiency and space. The ADIS protocol operations cost a certain amount in LISP function call and Tenex JSYS overhead; they also have a cost determined by the number of bytes of protocol commands that are sent to Chat. Thus we can express the communication cost in terms of the number of "characters" we could display by transmitting the same number of bits. Here are approximate numbers:

ADISRegion	4
ADISLimits	16
ADISSetX,ADISSetY,ADISFont	5

ADISBold,ADISItalic,ADISSetCR,ADISSetLF	5
ADISLineTo	6
ADISRegionOp	13 or 21
ADISScroll	34 in most cases
ADISButtonEnable	16
ADISTypeOnEvent	4
ADISCursor	43
ADISCursorMove	7

Space in the Alto is at a premium. At present, about 6700 words must be shared among all fonts and region descriptions. Note that font sizes vary. Sizes are:

Region	34 words (always)
Helvetica8.Strike	570 words
Helvetica10.Strike	630 words

CopyDisk

CopyDisk is a program for copying entire disk packs. It will copy from one drive to another on the same machine, or between drives on separate machines via a network.

1. History

The first Alto CopyDisk was called Quick and was written by Gene McDaniel in 1973. During the summer of 1975 Graeme Williams wrote a new CopyDisk adding the ability to copy disks over the network. During the summer of 1976 David Boggs redesigned the network protocol and added the ability to copy Trident disks. The CopyDisk network protocol is specified in <Pup>CopyDisk.cars.

2. Concepts and Terminology

In a disk copy operation, the information on a 'Source' disk is copied to a 'Destination' disk, destroying any previous information on the destination. A copy operation usually consists of two steps:

> [Copy] Step one copies bit-for-bit the information from the source disk to the destination disk.
>
> [Check] Step two reads the destination disk and checks that it is indentical with the source disk. This step can be omitted at the user's peril.

Copying a disk from one machine (or 'host') to another over a network requires the active cooperation of programs on both machines. In a typical scenario, a human user invokes a program called a 'CopyDisk User' and directs it to establish contact with a 'CopyDisk Server' on another machine. Once contact has been established, the CopyDisk User initiates requests and supplies parameters for the actual copy operation which the User and Server carry out together. The User and Server roles differ in that the CopyDisk User interacts with a human user (usually through some keyboard interpreter) and takes the initiative in User/Server interactions, whereas the CopyDisk Server plays a comparatively passive role. The question of which machine is the CopyDisk User and which is the CopyDisk Server is independent of the direction in which data moves.

The Alto CopyDisk subsystem contains both a CopyDisk User and a CopyDisk Server, running as independent processes. Therefore to copy a disk from one machine to another you should start up the CopyDisk subsystem on both machines and then type commands to one of them, which becomes the CopyDisk User. Subsequent operations are controlled entirely from the User end, with no human intervention required at the Server machine. This arrangement is similar to the way the Alto FTP subsystem works, and different from the way the older CopyDisk worked.

3. Calling CopyDisk

CopyDisk can be run in two modes: interactive mode in which commands come from the keyboard, and non-interactive mode in which commands come from the command line (Com.cm). The general form of the command line to invoke CopyDisk looks like:

CopyDisk [[/<option switches>] [from] <source> [to] <destination>]

The square brackets denote portions of the command line that are optional and may be omitted. If you just type "CopyDisk" the program goes into interactive mode, otherwise the remainder of the command line must be a complete description of the desired operation.

3.1. Option Switches

Each option switch has a default value which is used if the switch is not explicitly set. To set a switch to 'false' proceed it with a 'minus' sign (thus CopyDisk/-C means 'no checking'). To set a switch to 'true' just mention the switch.

Switch	Default	Function
/4	false	[Model44] tells CopyDisk to copy an entire Diablo model 44, without asking for confirmation.
/C	true	[Check] tells CopyDisk whether to check the copy operation. CopyDisk/-C, which omits the check step, is faster but more risky.
/W	true	[WriteProtect] prevents the CopyDisk network Server from writing on a local disk. So unless you say CopyDisk/W or issue the WRITEPROTECT command, someone can make a copy of your disk over the network, but no one can (maliciously or accidentally) overwrite it.
/R	true	[Ram] tells CopyDisk to attempt to load the ram with some microcode which speeds things up considerably. CopyDisk will still work, though more slowly if it can't load the ram.
/D	false	[Debug] enables extra printout that should be interesting only to CopyDisk maintainers.
/B	false	[Boot] creates 'CopyDisk.boot' for distribution to boot servers.
/A	false	[AllocatorDebug] enables extra consistancy checks in the free storage allocator.

3.2. Source and Destination Syntax

The general form of a source or destination disk name is:

[Host name]Device

for example "[Myrddin]DP0". Ordinarily 'host name' can be a string, e.g., "Myrddin". Most Altos have names which are registered in Name Lookup Servers. So long as a name lookup server is available, CopyDisk is able to obtain the information necessary to translate a host name to an inter-network address (which is what the underlying network mechanism uses). You may omit the host name for disks attached to the local machine.

If the host name of the Server machine is not known, you may specify an inter-network address in its place. The general form of an inter-network address is:

<network> # <host> # <socket>

where each of the three fields is an octal number. The <network> number designates the network to which the Server host is connected (which may be different from the one to which the User host is connected); this (along with the "#" that follows it) may be omitted if the Server and User are known to be connected to the same network. The <host> number designates the Server host's address on <network>. The <socket> number designates the actual Server process on that host; ordinarily it should be omitted, since the default is the regular CopyDisk server socket. Hence to specify a CopyDisk server running in Alto host number 241 on the directly connected network, you should say "241#" (the trailing "#" is required).

The syntax of the 'device' part of a disk name depends on the disk type. CopyDisk currently knows how to copy two kinds of disks:

DPn	Diablo disk unit 'n'. Most Altos have one Diablo disk called 'DP0'.
TPn	Trident disk unit 'n'. The unit number must be in the range 0-7.

4. The CopyDisk display

CopyDisk displays a title line about one inch from the top of the screen, and below that the main display window, which consumes about half of the screen. The main window is shared by the User and Server processes, only one of which is active at any time. The process which currently owns the window identifies itself at the right side of the title line. The title also shows the release date of the program and the host number of the Alto. When a copy operation is in progress, the current disk address is displayed in the area above the title line.

When CopyDisk is started, the User is listening for commands from the keyboard and the Server is listening for connections from the network. If you start typing commands, the User takes over control of the main window ('User' appears near the right end of the title line), and your commands and their responses are displayed there. The Server refuses network connections while the User is active. If another CopyDisk program connects to the Server, the Server takes over control of the main window ('Server' appears near the right end of the title line), and the Server logs its activity there. The User ignores type-in (flashing the screen if any keys are typed) while the Server is active.

5. Keyboard Command Syntax

CopyDisk's interactive command interpreter presents a user interface very similar to that of the Alto FTP subsystem. The standard editing characters, command recognition features, and help facility (via "?") are available.

5.1. Keyboard Commands

COPY

Starts a dialog to gather the information for copying a disk. CopyDisk first asks for the name of the source disk by displaying "Copy from". If the disk is local, it makes sure it is ready; if the disk is on another machine, it opens a connection and asks the remote machine if the disk is ready. If you want to abort the connection attempt, hit the middle unmarked ('Chat') key. If the source disk is ready, CopyDisk prompts you for the destination disk by displaying "Copy to", and then checks that that disk is ready also. Next, it verifies that the disks are compatible, and depending on the disk type, may ask some questions about things peculiar to that disk (such as "Copy all of the model 44?"). Then CopyDisk asks you to confirm your intention to overwrite the destination disk. If you change your mind, type 'N' or <delete>. If you respond yes, CopyDisk will pause for a few seconds, ignoring the keyboard, and then ask you to confirm once again. Type-ahead does not work for this second confirmation. This is your last chance to look at the disks and make sure that you are not overwriting the wrong one. It happens! This feature was in the original CopyDisk, was left out of the second version, and is back in this third version by popular demand from the many people who made that fatal mistake.

QUIT

Terminates CopyDisk. One of three things happens:

The Alto Exec is restarted if DP0 is ready, and has not been written on, and if CopyDisk was not booted from the net.

DP0 is booted if it is ready but has been written on or if CopyDisk was booted from the net.

DMT is booted from the net if DP0 is not ready.

All of this is attempting to leave the Alto running something useful. If the disk in DP0 does not have an operating system on it when CopyDisk quits, the disk boot (option 2, above) will fail. This will not hurt the disk, it will just leave the Alto in a bad state. You will have to boot DMT manually.

CHECK
Toggles the switch which controls whether a disk is checked after copying. CopyDisk displays "on" if checking is now enabled, and "off" if it is now disabled.

DEBUG
Toggles the switch which controls the display of debugging information. The performance data presented at the end of this document is part of the debugging information; the network protocol interactions are displayed when this switch is set also.

WRITEPROTECT
Toggles the switch which allows the network Server to write on local disks. The default is that people can't overwrite your disk.

VERIFY
Verifies that two disks are identical. The dialog is very similar to the COPY command. Neither disk is ever written. This is useful to verify the health of your disk drive (but remember that it does not check the write logic).

6. Command Line Syntax

CopyDisk can also be controlled from the command line. If there is anything in the command line except "CopyDisk" and global switches, the command line interpreter is started instead of the interactive keyboard interpreter. Its operation is most easily explained by examples:

6.1. Command line examples

To copy DP0 to DP1:

CopyDisk from DP0 to DP1

Note that 'from' and 'to' are optional (though stongly recommended for clarity), and one or both may be omitted or abbreviated:

CopyDisk DP0 t DP1

is equivalent, though less obvious.

To copy the Basic non-programmer's disk from host 'Tape-Controller' (which is running CopyDisk) onto a disk in your own machine:

CopyDisk from [Tape-Controller]DP0 to DP0

or, equivalently:

CopyDisk from [3'#6'#]DP0 to DP0

The single quotes are necessary to keep the #s out of the clutches of the Alto Exec. The quotes are not needed when typing to the keyboard interpreter. Note that no spaces are allowed between the host name and the device name.

If the command line interpreter runs into trouble, it displays an error message and then starts the interactive interpreter.

7. Disk Errors

Disk errors are termed 'soft' or 'hard' depending on whether retrying the operation corrects the difficulty. If CopyDisk is still having trouble after many retries, it displays a message of the form "Hard error at DPn: cyl xxx hd y sec zz" in the main window and moves on.

Soft errors are not reported unless the debug switch is true, so as not to alarm users. Their frequency depends on several factors. Copying over the network will cause more soft errors then copying between two disks on the same machine. Alto IIs get many more errors then Alto Is.

During the Check pass, in addition to soft and hard errors, 'data compare' errors are also possible. A data compare error means that the corresponding sections of the source and destination disks are not identical. If any hard errors have been reported, then data compare errors are likely, otherwise getting data compare errors means that something is very wrong. You should suspect the Alto.

Hard errors and data compare errors are serious, and you should not trust the copied pack if any are reported. If the errors are on the source disk, try Scavenging it. Bear in mind that there is some variance in alignment among disk drives, and that a pack which reads fine on one drive may have trouble on another. Is the source disk in a different drive than where it is normally used? Before allowing the Scavenger to rewrite sectors, consider that the pack may be OK, but the drive it is in may be out of alignment. In this case, allowing the scavenger to rewrite the sectors is a bad idea. If the errors are on the destination disk, try the copy again, and then suspect the pack or the disk drive itself. If the destination pack was much colder than the temperature inside the drive, sectors written early in the copy pass may read incorrectly during the check pass. It's a good idea to wait a few minutes for the pack to reach normal operating temperature before using it.

8. Creating a new disk

Suppose you want to make a new disk by copying one of the 'Basic' disks. There are two major ways to do this:

> Find an Alto with two disk drives. These are relatively rare beasts. This method is called the 'double disk copy' method.
>
> Find two Altos, each with one drive, that are connected by a network. This should be relatively easy. This method is called the 'network copy' method.

Having decided on one of the above methods, you must now get CopyDisk running on the Alto(s). There are two major ways to do this:

> Start CopyDisk from a disk which has 'CopyDisk.run' on it.
>
> Boot CopyDisk over the network from a 'Boot Server'.

8.1. Starting CopyDisk from another Disk

If you do not have access to a Boot Server, you must start CopyDisk from a disk that has it on it. Put a disk with CopyDisk on it into the Alto and type "CopyDisk<return>". Then switch disks. BE CAREFUL!! People sometimes forget to switch disks at this point and accidentally copy the wrong one. This is why CopyDisk asks you to confirm your intentions so many times.

8.2. Booting Copydisk from the net

The best way to start CopyDisk is to boot it from the network. That way you are more likely to get the latest version, and you avoid the pitfall mentioned above. Of course, you must have network access to a Boot Server. Most Gateways have Boot Servers. If this method doesn't seem to work, you will have to fall back to starting CopyDisk from another disk.

Hold down the <BS> and <Quote> keys while pressing the boot button on the Alto. You must continue to hold down <BS> and <Quote> (but let go of the boot button!) until a small square appears in the middle of the screen. This can take up to 30 seconds, but usually happens in less than 5 seconds. You are now taking to the NetExec (see the documentation in the Subsystems manual if you are curious), and you should type "CopyDisk<return>". The screen will go blank, the little square will appear again (you don't have to hold down any keys this time), and soon CopyDisk should appear on the screen.

8.3. The Double-Disk Copy Method

Put the basic disk in DP0 and put your disk in DP1. Type "Copy<space>", and when it says "from" type DP0<return>. When it says "Copy to", type "DP1<return>". Then type <return> each time it asks for confirmation. Some numbers will appear in the top center of the screen. When they disappear, CopyDisk is done. Type "Quit<return>". Put the basic disk back where it belongs, and take your disk with you.

8.4. The Network Copy Method

Unlike the old CopyDisk, you need only type commands to one of the two Altos. It doesn't matter which one. Assume that the basic disk is in the Alto called "Tape-Controller", your disk is in the Alto called "Myrddin" and you are going to type commands to Tape-Controller. Type "Copy<space>", and when it says "from" type "DP0<return>". When it says "Copy to", type "[Myrddin]DP0<return>". Then type <return> each time it asks for confirmation. Some numbers will appear in the top center of the screen. When they disappear, CopyDisk is done. Type "Quit<return>", and put the basic disk back in the rack. Go to Myrddin and type "Quit<return>". It will boot the disk, and you should find yourself talking to the Alto Exec.

9. Performance

This section calculates the times to copy disks under different conditions. CopyDisk times its operations and displays the results if the debug switch is set, so you can compare the numbers derived here with reality.

9.1. TSweep

First, we calculate TSweep, the time to read or write a disk assuming that we can consume or produce data faster than the disk. This best possible case is the sum of two terms. The first term is the time necessary to sweep an active read/write head over every sector on the disk:

Rot * nCyl * nHds.

The second term is the time lost while seeking to the next cylinder. We assume that these seeks take less than one rotation but that a whole rotation is lost:

Rot * nCyl.

Combining, we get:

TSweep = Rot * nCyl * (nHds+1).

where: Rot is the rotation time of the disk in seconds
nCyl is the number of cylinders, and
nHds is the number of heads.

9.2. Disk-To-Disk Copy

By disk-to-disk copy we mean copying from one disk to another on the same machine, using a single controller and not overlapping seeks. The fastest way to do this is to read the entire source disk into memory, switch to the destination disk, and then write it all. The switch from the source to the destination disk, will lose on the average half a revolution while waiting for the right sector on the new disk to come under a head. Neglecting the switch time which is small compared to the other two terms, the best possible disk-to-disk copy time is 2 * TSweep.

With limited memory, the best we can do is fill all available memory buffers reading the source disk, switch disks, write them onto the destination disk, and then switch back to the source disk for another load. In this case we can't ignore the switch time, which is the total number of sectors on the disk divided by the number of sector buffers times the rotation time of the disk:

Rot * (nCyl * nHds * nSec)/nBuf

where nSec is the number of sectors per track, and
nBuf is the number of memory buffers.

So the disk-to-disk copy time, TDDCopy, is:

TDDCopy = 2 * TSweep + Rot * (nCyl * nHds * nSec)/nBuf

9.3. Net Copy

By net copy we mean copying from a disk on one machine through a network to a disk on another machine. In this case the disk controllers can be going in parallel, and the factor of two in the first term of TDDCopy vanishes. In additon, if the bandwidth of the network connection is higher than the transfer rate of the disks so that as soon as a sector is read from the disk it is sent out of the machine, the memory limitation goes away and the second term of TDDCopy vanishes.

The CopyDisk network protocol sends a small amount of information along with each sector which must be factored into the calculation of the bandwidth needed to run without memory limitation. Note that the bandwidth we are concerned with here is that perceived by a client of the network services: user data bits per second, not raw bits per second through the network hardware.

If the network is slower than the disks, then the time to copy a disk is the time required to transmit all of the bits on a disk plus the protocol overhead bits:

TNetCopy = nCyl * nHds * nSec * (sB + sOv)/bwNet

where sB is the bits of disk information per sector,
sOv is the CopyDisk protocol overhead per sector, and
bwNet is the bandwidth of the network connection.

The bandwidth of the network connection is hard to state, and depends on a number of factors. Here are a few:

Reduction of the emulator's instruction execution rate due to interference from the disk and network hardware.

Reduction of the amount of the emulator cycles available to the network and disk code due to mutual interference.

Reduction of the peak network bandwith due to interference from other hosts on the network.

The minimum network bandwith required to copy a disk at full speed is:

$$MinBwNet = 16 * nCyl * nHds * nSec * (sB + sOv)/TSweep.$$

9.4. The Numbers for Altos

Here are the relevant numbers for the disks which this program can copy:

	Diablo-31	Diablo-44	Trident-80	Trident-300
Rot (ms)	40	25	16.66	16.66
nCyl	203	406	815	815
nHds	2	2	5	19
nSec	12	12	9	9
sB	266	266	1036	1036
sOv	2	2	2	2
nBuf	80	80	18	18

9.5. Reality

Here are the results of plugging the numbers into the equations, and comparing them against actual measurements. The format is predicted(measured). NA means not available.

	Diablo-31	Diablo-44	Trident-80	Trident-300
TSweep	0:24	0:30	1:21	4:32
TDDCopy	0:51(0:51)	1:04(1:16)	3:18(4:00)	11:20(19:27)
TNetCopy	(1:05)	(2:16)	(26:31)	(NA)
bwNet	(323 Kb/s)	(308 Kb/s)	(383 Kb/s)	(NA)
MinBwNet	859 Kb/s	1.375 Mb/s	7.520 Mb/s	8.509 Mb/s

10. Revision History

August 7, 1977

First relese.

August 28, 1977

Soft errors are only reported if the debug switch is set. Data compare errors now display the offending disk address. VERIFY and WRITEPROTECT commands added to keyboard command interpreter. Write protect global switch added.

October 16, 1977

More microcode to speed things up

October 27, 1977

Bug fixes

December 18, 1977

Fixed a bug which prevented it from copying the second half of a two disk file system. The network format for Diablo disks changed.

March 22, 1978

CopyDisk will now do the right thing for "[thisHost]device". The default value of WRITEPROTECT is now TRUE.

Createfile

This subsystem creates a file of a given size, attempting to allocate it contiguously on the disk. To run the program, use

>CreateFile filename npages

where filename is the name of the file and npages is the size of the file in pages (in octal unless you suffix a "d": 99d). This program is primarily intended for creating files which will be accessed using the Indexed Sequential File (ISF) package, which influences its notion of what a contiguous file looks like. The algorithm is: 1) search the disk bit table and locate the largest group of contiguous free pages. 2) if nPages is less than the size of this group, allocate nPages and finish; otherwise allocate the whole group, decrease nPages by the size of the group and repeat step 1. This program can be fooled into allocating pages in less than optimal ways if your bit table is not in sync with the disk, so if in doubt, run the Scavenger first. If there aren't enough pages on your disk, it will fail gracefully, perhaps after thrashing around for a while.

DDS - Descriptive Directory System - release 1.13

The Descriptive Directory System (DDS) is a front end for the Alto file system that provides substantially greater flexibility than the "?" facility in the operating system's command processor. In addition to file names, the DDS can display file lengths, creation-read-write dates, and contents.

If you have used DDS before and merely wish to learn about changes, bug fixes, and new features, you probably want to skip to section 5 of this document. If not, sections 0 through 4 are a complete description of the current release. Sections significantly changed since the last release are marked with ***.

0. The mouse and cursor

The three buttons on the mouse are called RED (left or top button), YELLOW (middle button), and BLUE (right or bottom button). Most mouse-controlled actions in DDS happen as soon as you depress the mouse button: these are described below using phrases like "RED does xxx", meaning "As soon as you depress RED, xxx happens." Some actions require depressing a button and then releasing it: phrases like "clicking RED does xxx" mean "If you depress RED and then release it, xxx will happen." Careful reading, or a little experimenting, will familiarize you quickly with the distinction.

The cursor changes shape according to its location on the display and according to how DDS is interpreting the buttons. Generally speaking, when the cursor is circular, RED selects what you are pointing at in some way, and BLUE deselects it. When the cursor assumes the shape of an hourglass, DDS is busy doing something and is not listening to the mouse buttons.

1. The display

Like Bravo, DDS divides the display into a command area at the top, and one or more windows below. Currently DDS just supports a single window. A heavy black bar separates the command area from the window. Section 2 (below) describes the command area.

The window has three parts, separated by lighter horizontal bars:
1) The top part is the view specification area, or viewspec area for short. It contains a set of keywords that describe what information is to be displayed for the files being examined in this window, and a set of keywords that describe how the displayed files are sorted.
2) The second part is the selection specification area, or selspec area for short. It contains a pair of expressions which together determine what set of files is being examined in the window. View and selection specification are completely independent: each can be changed without affecting the other.
3) The main part of the window is the data area, which actually displays a set of files. The names are always displayed: other information is controlled by the viewspecs.

1.1 The viewspec area

There are 10 keywords in the viewspec area that control what is displayed:
"created" - the date when the file was created
"written" - the date when the file was last altered
"read" - the date when the file was last read
"referenced" - the date when the file was last referenced (i.e. the most recent of "created", "written", and "read")
"size" - the size of the file in disk pages
"length" - the length of the file in bytes (characters)
"address" - the hardware address, in the form directory-pointer: (SN1,SN2)!VN @ virtual-leader-address
"contents" - the contents of the file (in octal, if a binary file)
"pagemap" - the disk addresses of all pages of the file, with a "*" before each address that represents a change of head position
"leader" - the contents of the file's leader page, in octal

If the keyword is displayed white-on-black, the corresponding information is displayed in the data area, otherwise not.

There are 6 keywords that control other aspects of how the data are displayed:
"(marked)" - if turned on, DDS only displays marked files (see sec. 1.4 below)
"(small)" - if turned on, DDS uses a smaller font for the data, which allows more data to appear on the screen (see sec. 3 below for how to tell DDS the name of the font)
"(packed)" - if turned on, DDS displays several files per line if possible (not implemented yet)
"(times)" - in conjunction with "created", "written", "read", or "referenced", shows the time of day as well as the date
"(browse)" - if turned on, then when "contents" is turned on, DDS only displays the first 5 lines of text files and the message "*** binary file ***" for binary files, instead of the complete contents of the file.
"(chart)" - if turned on, changes the data display to be a chart made up of boxes in which the height of the box is proportional to the file length. (Try it -- you'll like it.)

When the cursor is positioned over a keyword name, RED turns the keyword on; BLUE turns the keyword off. When the cursor is over the word "Show:" at the upper left of the keywords, BLUE turns all keywords off.

There are currently 8 keywords that control sorting of the data:
"name" - alphabetic order by name (upper and lower case letters are equivalent)
"extension" - alphabetic order by extension
"created", "written", "read" - the corresponding date and time
"referenced" - the date last referenced
"length" - the file length
"serial" - the file's serial number (not of general interest)

The keywords which are displayed white-on-black are those actually used to sort the data area. They are displayed in the order most- to least-significant criterion, e.g. "extension↑" followed by "name↑" means sort by extension first, then sort files with the same extension by name. Following each keyword, whether active or not, is an arrow which indicates whether the sort is to be in ascending (upward arrow) or descending (downward arrow).

When the cursor is positioned over a sorting keyword name, clicking RED turns the keyword on and adds it to the list of white-on-black keywords actually used for sorting; clicking BLUE turns the keyword off and removes it from the list; clicking YELLOW inverts the direction of the arrow, regardless of whether the keyword is in the list. When the cursor is over the words "Sort by:" at the left of the sorting keywords, BLUE turns off all sorting criteria.

Since sorting may take a long time and it is easy to request sorting by accident, you can abort sorting at any time by typing any character. Be sure the cursor is not in the data area when you do this: if it is, DDS may start the sort over again!

Whenever the cursor moves into the data area, regardless of whether any mouse buttons are down, DDS repaints the display to be as specified by the viewspecs if the viewspecs have changed since the last time the display was repainted.

1.2 The selspec area

The selspec area contains two expressions which defines what subset of the directory will actually be displayed in the data area. These expressions are built up from name patterns which are similar to those recognized by the Alto Executive. More precisely, a name pattern is a sequence of characters which may contain "*"s and "#"s: "*" matches any sequence of characters in a name (including no characters at all), "#" matches any single character. Upper and lower case letters are not distinguished. Note that DDS deletes the final "." from file names. Here are some examples of name patterns and what they match:

*.BC All files with extension BC (or bc, bC, or Bc).
*.B All files with extension B.
.B All files whose names contain the string .B -- this includes all files with extension Bsomething, but also includes files like THIS.BINARY.THAT.
*.B# All files whose extensions consist of B and one more character.

* All the files in the directory.

You can build up more complex expressions using the words "and", "or", and "not", and parentheses. Here are some examples of such expressions and what they select:

LPD* and not *.temp
All files beginning with LPD, except those with extension temp.

*.memo or *.memo$
All files with extension memo or memo$.

(*.BT or *.BS) and not X*
All files with extension BT or BS, except those beginning with X.

The upper expression in the selspec area is called the selspec; the lower one is called the context. (The two together are simply called the selspecs.) Only files which satisfy both expressions will be displayed. The idea is that if you are going to be working on memos, for example, you can set the context to "*.memo" and use the selspec to further select within this set. As another example, if there is some set of files you want not to see (like "*$"), you can set the context to "not *$".

To change the selspec or the context, point at it, or at the word "Selspec:" or "Context:", and click RED or YELLOW. This will cause it to change to white-on-black. As soon as you start typing, the old text will vanish and what you type will appear white-on-black in its place. Eventually you must type one of the following three things before you can point anywhere else or select any commands (see sec. 2 below):

<return> confirms the change, and repaints the display to reflect it.
<esc> is equivalent to *<return>, i.e. it adds a * to what you have typed and then confirms the change.
<del> aborts the typein and restores the old selspec or context expression.

See section 3 below for how to get the selspec and/or context initialized automatically to something other than "*" when you first enter DDS.

The third line of the selspec area is a message of the form "nnn files are selected, of which mmm are marked" where nnn is the count of files selected by the current selspec and mmm is the count of those which are marked (see 1.4 below). If there are marked files not selected by the selspec (again, see 1.4), the message "there are kkk files marked but not selected" also appears. While DDS is sorting data, the message "Sorting ..." appears in this area in place of the file counts.

1.3 The data area

As mentioned above, whenever the cursor moves into the data area, DDS regenerates the display if necessary to conform to the current viewspecs.

The left edge of the data area is a scrolling bar which works the same way as in Bravo: clicking RED scrolls up, clicking BLUE scrolls down, and clicking YELLOW jumps proportionately to the vertical location in the window. A hollow arrow in the left margin shows where in the list you are positioned: if the arrow is at the top, you are at the beginning of the list; if the arrow is at the bottom, you are at the end. The idea is that if you were to move the cursor to this arrow and click YELLOW, the list would stay positioned just as it is. (This feature may appear in Bravo some day too.)

If you are positioned at the beginning of the list of selected files, DDS displays the message "~~~~~ BEGIN ~~~~~" at the head of the list; if not, DDS displays "~~~~~ nnn files not shown ~~~~~", indicating the position within the list of the first file actually shown on the screen (e.g. "2 files not shown" means the first file on the screen is actually the third in the list). Similarly, if the last file shown on the screen is actually the last file in the list, DDS displays "~~~~~ END ~~~~~" below it.

A vertical strip at the right edge of the data area will be used in the future to control the formatting of the screen into windows. Currently the cursor changes shape when it is in this area, but the buttons have no effect. Another vertical strip just to the left of this one is used for mass marking and unmarking of files: see the following section.

1.4 Marking files

DDS provides a facility for marking any set of files for later processing by commands such as <Delete>, <Send to Maxc>, etc. Marked files are displayed with a small dark arrow in the left margin, and a count of how many marked files are in the current selected set is maintained in the selspec window. When the cursor is in the data area of a window, other than the right or left edge areas, the mouse buttons control marking and unmarking of individual files: RED marks the file on whose line the cursor resides; BLUE unmarks the file. When the cursor is in the vertical strip about 1" in from the right edge of the screen, the cursor changes to the word "ALL", and the buttons mark and unmark files en masse: clicking RED marks all the files selected by the selspecs; clicking BLUE unmarks all the files.

Note that files may be marked even though they are not selected by the current selspecs, i.e. marking is associated with the file rather than the display. (If this proves confusing it will be changed.) The count of "files marked but not selected" in the selspec area lets you know when there are marked files not selected by the current selspecs.

Since marking or unmarking individual files occurs as soon as the button is depressed, you can hold down RED or BLUE and slide the mouse (slowly) in the vertical direction to mark or unmark a group of adjacent files.

The marked file counts in the selspec window are adjusted as soon as a file is marked or unmarked, but if the "marked" viewspec is on and you unmark a file, you must scroll the data to get the unmarked file(s) deleted from the display.

2. Commands

The command area at the top of the screen consists of four parts:
1) A header with the DDS version number, time of day, and count of free disk pages;
2) A type-in area, where typed characters appear;
4) An error message line;
3) A menu of commands, with each built-in command being enclosed in angle brackets <>.

When the mouse is in the command menu area, RED selects a command for subsequent execution: the selected command is displayed white-on-black, and any previously selected command is deselected. BLUE deselects the currently selected command and selects the default command <Quit>. Typing <esc> or <return> finally initiates the command: you can freely select or deselect commands, type and edit your type-in, change viewspecs, etc. up to that moment. For commands which do not require type-in, you may also initiate the command by clicking YELLOW with the mouse in the command menu area. The cursor takes the shape of a circle with a cross when this is allowed, and a circle with a dot when it is not.

Some commands require or allow type-in before the final <esc> or <return>. You may type at any time. All typed characters are accumulated in the type-in area just below the header until the <esc> or <return>. Control-A (or backspace), control-W, control-Q, and <del> are available for editing as in Bravo. DDS displays a vertical bar when it is waiting for your typing, and of course you can "type ahead" while DDS is processing a command. However, as for selspec and context changes (sec. 1.2), once you have started to type, you must either confirm the command with <esc> or <return>, or abort with <del>, before you can select another command or another place to type (selspec or context).

When you have selected a command with RED, then when you release the button, DDS may display something in the type-in area which is a default for that command. If you want to execute the command with that default type-in, you can just confirm it (with <esc>, <return>, or YELLOW); otherwise, the default disappears as soon as you start typing, just like the old selspec or context.

In the description of commands below, "something" following the command name means that DDS expects you to have typed something before the final <esc> or <return> that initiates the command; "optional-something" means you may type something or not. To help you remember, all the commands that require type-in end with "...", and those which allow but do not require type-in end with "[...]".

Many commands operate on a set of files: they use precisely those files which, at the time you type the final <esc> or <return>, are both selected (i.e. match the selspec) and marked. "Filename-1 ... filename-n" in the descriptions below refer to these files, which are also called the "designated" files.

DDS presently has two classes of commands: those which leave you in DDS after execution (internal commands), and those which send you back to the Alto Executive (external commands). DDS has a fixed collection of internal commands, but you can add new external commands of your own: see section 3 below for how to do this. For external commands, DDS saves away a command line so that if something goes wrong, you can execute the command again by typing @DDS.CM@<return> to the Executive.

2.1 Internal commands (those which leave you in DDS)

<Put on file ...> "filename" writes on the file named "filename" (in text form) the contents of the window. DDS also writes a header with your name, the disk name, and the date and time. The default for "filename" is "Dir.Lst", an arbitrary name which DDS supplies so that you don't have to make one up.

<List on file ...> "filename" writes on the file named "filename" (in text form) the names of the designated files, separated by blanks. This makes it easy for you to make up an @-file for the Executive by adding a command name to the front of this file. The default for "filename" is "Dir.Cm", an arbitrary name which DDS supplies so you don't have to make one up.

<Delete> deletes the designated files. There is presently no way to un-delete files, so be careful: the count of marked files in the selspec window is a good clue as to whether you are deleting more than you want. You can stop a <Delete> at any time by typing any character: of course, some files may already have been deleted. DDS changes the "free pages" count at the top of the screen as it deletes each file.

<Rename as ...> "filename" requires that there be exactly one designated file, and changes its name to "filename". If there is already a file named "filename", <Rename> gives an error message and does nothing else.

<Initialize [select ...]> "selspec" restores your selspec, context, and viewspecs to what you have specified in User.Cm. If you typed something, DDS takes that in place of the selspec in User.Cm.

2.2 External commands (those which leave you in the Executive)

<Quit> leaves DDS gracefully. Shift-Swat is also safe whenever DDS is awaiting input (i.e. not in the midst of sorting, deleting, etc.).

<Bravo/[...]> "optional-switches" gives control to Bravo in the following way:
If there are no designated files, DDS effectively executes "Bravo/switches".
If there is more than one designated file, DDS gives an error message and does nothing else.
If there is a single designated file and you did not type anything, DDS effectively executes "Bravo/N filename", i.e. instructs Bravo to read in the file.
If there is a single designated file and you did type in switches, DDS executes "Bravo/switches filename".

<Gears/[...]> "optional-switches" executes "Gears/switches filename-1 ... filename-n", i.e. prints the designated files.

<Send to Maxc directory <...>> "directory-name" sends the designated files to the directory named "directory-name" on Maxc, using Ftp. The default for directory-name is the user name on your Alto disk. If you accept the default, DDS assumes you have already done a Login in the Executive to supply the password; if you supply some other directory-name XYZ, DDS arranges things so the Executive will prompt you with the message "File XYZ-Password does not exist, type what it would contain" and you should type in the password for XYZ at that time.

<Send to ...> "name" sends the designated files to the Alto whose name is "name", using Ftp. "Name" may be anything acceptable to Ftp, i.e. an Alto name, an Alto number, etc. The default for "name" is Maxc, which is not really very useful.

<Execute ...> "command" constructs a command line formed from "command" and the names of the designated files, and then executes the command line thus formed by either jumping directly to the subsystem or returning to the Alto Executive. (If there are no designated files, DDS produces an error message "No files are marked" and does nothing else.) The command line is formed in the following way:

If "command" does not contain any "*" characters, the command line is just "command" followed by the names of the designated files. For example, if files ALPHA and BETA are designated, <Execute ...> "BLDR/L" would execute the command line "BLDR/L ALPHA BETA". "String" may contain blanks, so for example <Execute> "BLDR FOO/S" would execute "BLDR FOO/S ALPHA BETA".

If "command" does contain a "*", DDS divides "command" into 3 parts "s1 s2*s3", where s2 is the part of "command" extending backwards from the "*" to the first preceding blank (or the beginning of "command"). Then the command line is "s1 s2f1s3 s2f2s3 ..." where f1, f2, etc. are the names of the files. For example, if ALPHA and BETA are designated, <Execute ...> "BLDR @*@" would execute the command line "BLDR @ALPHA@ @BETA@". (If this seems confusing or useless, don't worry about it too much -- some future version of DDS may find a better way to provide this facility.)

2.3 User-defined commands

If you define your own external commands with a SUBSYSTEMS entry in User.Cm as described in section 3 below, these commands will also appear in the command menu along with all the commands listed just above. They behave exactly like the <Execute> command with respect to what they do about *'s, typein, and designated files. For example, suppose your SUBSYSTEMS list looks like this:

SUBSYSTEMS: Chat, Ftp/-S Maxc, Foo

Then if you select the second command with files Alpha and Beta designated and type Dump/C Blap.DM, what will actually get executed is Ftp/-S Maxc Dump/C Blap.DM Alpha Beta.

2.4 Error messages

Non-fatal error messages appear in bold characters just below the type-in line. Such messages always abort the current command and reset the command to <Quit>, but they do not change the state of DDS in any other way. The message disappears as soon as you type any character.

Fatal errors cause DDS to call Swat. When this happens, the screen changes completely and a heading like "Swat.21 (August 28, 1976)" appears at the top; the error message itself appears at the bottom of the screen just above a "#". Fatal errors are never supposed to happen, but if one ever does, summon a DDS expert. If none is available, write down the message and what you were doing at the time, and then type control-K. This will throw you out of DDS and back to the Executive.

3. User profile

DDS examines the user profile (User.Cm) during initialization to obtain the names of the fonts which will be used to display various things, and other rarely-changed information. Just as Bravo's section of User.Cm begins with [BRAVO] and then follows the format of OPTION:STRING, DDS looks for [DDS] and follows the same format for its entries.

The entries which DDS recognizes in User.Cm fall into two classes. "Initialization-only" entries are those which DDS only consults when you ask it to do a full initialization (by using the FULLINIT: Yes entry in User.Cm, or the /I switch in the command line, both described below). "Ordinary" entries are those which DDS looks at every time.

The names of the "ordinary" entries are:

FONT: fontname - specifies the name of the normal font (used for the command window, the file count line, and the data area).

BOLDFONT: fontname - specifies the name of the bold font (used for error messages, the viewspec and selspec display, and the headings on the data area).

SMALLFONT: fontname - specifies the name of the small font (used for displaying data when the "(small)" viewspec is turned on).

SMALLBOLDFONT: fontname - specifies the name of the small bold font.

USERTYPE: type - lets DDS know what kind of user you are. If type is NON-PROGRAMMER, DDS doesn't provide the "pagemap" and "address" viewspecs. If type is WIZARD, DDS provides some extra features for debugging which are not described in this document.

WINDOWS: Yes - enables you to use some experimental facilities for splitting the screen into multiple windows in a Bravo-like manner. These facilities are NOT DOCUMENTED, NOT FULLY DEBUGGED, AND NOT RECOMMENDED.

RAMOK: Yes - tells DDS to use the RAM on your Alto. If your Alto is a standard one, this will make DDS run about 30% faster; if not, DDS may not run faster, and may not run at all. Try it once (or use the /R switch in the command line as described below) and see what happens.
FULLINIT: Yes - tells DDS to scan the whole Alto file directory each time it starts up, and reinitialize the selspec, context, etc. from the "initialization-only" entries in User.Cm (possibly overridden by the command line: see sec. 4). FULLINIT: No - tells DDS to update its knowledge of the world from Sys.Log (an incremental record of file activity since you last ran DDS), and restore the selspec, context, etc. to what they were when you last left DDS. The default is FULLINIT: No which leads to much faster startup. BECAUSE OF DEFICIENCIES IN THE ALTO OS AND IN BRAVO, THE RELEASED VERSION OF DDS FORCES FULLINIT: YES REGARDLESS OF WHAT IS IN USER.CM.
REENTER: Yes - tells DDS that you want to go back to DDS after completion of an external command. (Normally the Executive retains control after an external command finishes.)

The names of the "initialization-only" entries are:
SELSPEC: expression - specifies the initial value of the selspec when you enter DDS. If there is something illegal about the expression, DDS just uses "*" for the initial selspec, as though there were no SELSPEC entry in User.Cm.
CONTEXT: expression - specifies the initial value of the context when you enter DDS.
SHOW: list of viewspecs - allows you to initialize the viewspecs. Use commas between viewspecs if there is more than one.
SORT BY: list of sorting keywords - allows you to initialize the sorting order. Each keyword may be followed by "↑" for ascending order or "←" for descending order (neither means ascending order). Use commas between keywords if there is more than one.
SUBSYSTEMS: list of commands - allows you to add your own favorite subsystems to DDS' command set. Each command may be just a subsystem name (e.g. Chat) or a subsystem name followed by some initial arguments (e.g. Ftp/-S Maxc Dump/C). Use commas between entries if there is more than one.

A word about fonts: if FONT is not specified in User.Cm, DDS uses the standard system font SysFont.AI. If BOLDFONT is not specified, DDS fabricates a boldface version of the normal font, whatever it may be. If SMALLFONT is not specified, the "(small)" viewspec has no effect. If you specify a font name and there is no file by that name, DDS just ignores that entry in User.Cm.

4. The command line

Just typing DDS to the Alto Executive will activate DDS in its normal way, in which various aspects of its behavior are controlled by entries in User.Cm if present. However, you can override User.Cm by typing switches following the name DDS to the Executive. Here are the switches currently implemented:
DDS/E - equivalent to REENTER: Yes in User.Cm.
DDS/-E - overrides (cancels) REENTER: Yes in User.Cm.
DDS/I - equivalent to FULLINIT: Yes in User.Cm.
DDS/-I - overrides (cancels) FULLINIT: Yes in User.Cm.
DDS/R - equivalent to RAMOK: Yes in User.Cm.
DDS/-R - overrides (cancels) RAMOK: Yes in User.Cm.
DDS/W - equivalent to WINDOWS: Yes in User.Cm.
DDS/-W - overrides (cancels) WINDOWS: Yes in User.Cm.
DDS/S - causes DDS to write some statistics in a file DDS.STATS. Not currently of general interest.
DDS/P - causes DDS to write some data regarding disk activity in DDS.STATS. Not of general interest.
DDS/X - causes DDS to display some mysterious statistics at the top of the screen. Not of general interest.
These switches can be combined, e.g. DDS/I/R causes both full initialization and use of the RAM. Switches may be either upper or lower case.

If DDS is doing a full initialization (either because FULLINIT: Yes appears in User.Cm or because you said DDS/I), you may also supply initial selspec and context strings in the command line, and these will take precedence over those in User.Cm, if any. Unfortunately, the Alto Executive makes it a little inconvenient to include *'s in these strings, and you can't have blanks in them at all. To include a *, you must type '*, e.g. to start up DDS and specify alpha* as the selspec, you must type
DDS/I alpha'*

to the Executive. To specify beta* as the selspec and *.cm as the context, you must type
DDS/I beta'* '*.cm

5. Record of bug fixes, changes, and enhancements

Release 1.13:

Bugs fixed: user-defined commands were usually ignored even on full init.

Additions: REENTER in User.Cm (sec. 3); /E in command line (sec. 4).

Release 1.12:

Bugs fixed: crash if User.Cm!n existed but no User.Cm.

Changes: fast startup permanently disabled.

Additions: "leader" viewspec (sec. 1.1); <List> and <Initialize> commands (sec. 2.1); user-defined commands (sec. 2.3, 3); /X in command line (sec. 4).

Release 1.11:

Bugs fixed: falling into Swat when running on non-standard Alto configurations; fast startup now works.

Changes: can point at "Selspec:" and "Context:" (sec. 1.2); feedback after deleting each file (sec. 2.1); user and disk name appear on <Put> file (sec. 2.1); fast startup is the default (sec. 3).

Additions: WINDOWS and RAMOK in User.Cm (sec. 3); switches, initial selspec and context in command line (sec. 4).

Release 1.10:

Bugs fixed: "Bad VP" and "Bad tree" from <Delete>.

Changes: runs only under Alto OS version 5 or later; typing in selspec directly (sec. 1.2), "All" strip for marking/unmarking all files (sec. 1.3, 1.4), new typein scheme for commands (sec. 2); change in <Send> commands (sec. 2.1).

Additions: "(chart)" viewspec for pictorial file lengths (sec. 1.1); BEGIN, END, arrow for clearer indication of position within data list (sec. 1.3); default typein for commands (sec. 2); saving command line in DDS.CM (sec. 2); initializing viewspecs and sorting from User.Cm (sec. 3); fast startup feature (sec. 3).

Release 1.9:

*** There was no official release 1.9.

Release 1.8:

Bugs fixed: stack overflows (really!), "Vstream error" after <Delete>; file name from <Put> wasn't getting added to data base.

Changes: runs under new Alto Operating System; "contents" viewspec shows the whole file (sec. 1.1); marking all files is now done in selspec area (sec. 1.4); error message line moved to just below type-in line (sec. 2).

Enhancements: "referenced", "(browse)", and "(small)" viewspecs (sec. 1.1); interrupting sorting by

typing (sec. 1.1); context expression (sec. 1.2); initiating commands with YELLOW in command menu (sec. 2); <Context> and <Rename> commands (sec. 2.1); interrupting <Delete> by typing (sec. 2.1); SMALLFONT, SMALLBOLDFONT, SELSPEC, CONTEXT, USERTYPE options in User.Cm (sec. 3).

Release 1.7:

Bugs fixed: "Break at 0" or "Break at 1" during <Delete>; occasional stack overflows ("Break at getframe+36").

Changes: error messages now appear in their own area (sec. 2.2); cursor need not be in the window when confirming a command (sec. 2).

Enhancements: documentation sec. 2 has been expanded and improved to clarify the notion of designated files.

Release 1.6:

Bugs fixed: DDS would go into SWAT "Break at getframe+36" (stack overflows); also occasional "Bad vp" or "Vstream error" messages. A couple of typos in the documentation also fixed.

Enhancements: blinking caret for type-in (sec. 2); complex selspec expressions (sec. 1.2); count of marked files not selected (sec. 1.2, 1.4).

Release 1.5:

Changes: command menu in place of control characters (sec. 2); viewspecs do not require clicking (sec. 1.1).

Enhancements: Delete, Send, Bravo, Gears commands are built in (sec. 2); sorting by serial # (sec. 1.2).

Release 1.4:

Changes: date-and-time line rearranged; better behavior when displayed properties do not fit on one line.

Enhancements: "Sorting ..." message (sec. 1.2); "*" feature in ↑Execute (sec. 2).

Release 1.3:

Bugs fixed: system would blow up on any attempt to produce an error message such as "Mouse is not in a window"; system would sometimes blow up when starting up; the date-and-time line no longer blinks.

Changes: ↑Execute now only processes marked files (sec. 1.4, 2); sorting by extension is implemented (sec. 1.1).

Enhancements: marking individual files (sec. 1.4); displaying the file count (sec. 1.2, 1.4); "pagemap" viewspec (sec. 1.1); user-selectable fonts (sec. 2.1).

DMT, Peek, PeekSum

This documentation describes the operation of three related Alto Subsystems: DMT, the Memory/Control Ram diagnostic; Peek, the program to which DMT reports its findings; and PeekSum, the program which summarizes the reports collected by Peek.

1. Creating a Peek Disk

You should devote a separate disk to Peek. Boot files can take up a lot of space and the Peek report file can get quite large over a long holiday weekend if your network has many hosts. To avoid coming in on Monday and discovering your Peeker in Swat out of disk space, clean the disk out regularly. Peek automatically keeps its network directory and boot files up-to-date, so building a new peek disk amounts to building an bare disk (OS, Exec, Ftp, Empress, perhaps Bravo), getting Peek and PeekSum and just running it: it does the rest. I have written a canned procedure for building a Peek disk from scratch:

1) Boot an OS from the net and respond 'Yes' when it asks if you want the long installation dialog, and 'Yes' when it asks if you want to ERASE the disk.

2) When the erase procedure finishes, retrieve [Maxc]<Alto>PeekDisk.cm and invoke it by typing to the Exec:
>@PeekDisk.cm@

3) When the smoke clears, install your printer's name in the [HardCopy] section of user.cm and re-install Bravo. If you aren't on the west coast, change the ZONE parameter (e.g to +5:00 if you are on the east coast).

2. History

Chuck Thacker made DMT (early 1973) by combining many small diagnostics which he had developed to stress main memory using certain emulator instructions. There were originally two versions: PMT (Printer Memory Test) which logged statistics on the Diablo printer; and DMT (Display Memory Test) which used the display. Later (late 1973), an Ethernet driver was added to DMT. Bob Metcalfe wrote Peek, and Chuck wrote PeekSum. At this point, development and maintenance of PMT stopped. Still later (mid 1975), David Boggs added a Control Ram test to DMT, rewrote the Ethernet driver and took over maintenance. Nate Tobol, who designed the Alto II memory system, wrote the Alto II memory test (mid 1976) which was merged into DMT. David rewrote Peek and took over its maintenance. Doug Clark extended PeekSum, and took over its maintenance (early 1977).

3. DMT

DMT is written in the Alto BCPL-compatible variant of machine language and is distributed as a type-B boot file (see the BuildBoot documentation for more details).

When DMT is running, the Alto screen is black with a white cursor changing position once each time through the main loop. For Alto I the cursor flips at random intervals; for Alto II the interval is about 1 second. On Alto IIs with extended memory, the cursor contains a number between 0 and 3 indicating which bank it is currently testing. DMT contains a TeleSwat server. The key combination <Control><Left-Shift><Swat> causes DMT to stop and enter the debugger.

3.1. Statistics

If the 'S' key is depressed, DMT will display (and transmit on the Ethernet) the statistics it has accumulated. The display looks something like this:

DMT of 25 Dec 78, Alto II XM 241. 456 blocks, testing 17341 to 176777
0 bad main memory chips
0 bad control memory chips

If there are errors, a line describing each type of error will be displayed, and then, if the errors can be resolved to a particular chip, the Card, Row and Column (for Alto I), or the Card and Chip number (for Alto II) will be displayed. This display will stay up as long as the "S" key is depressed. Periodically the statistics are automatically broadcast on the Ethernet and appear briefly on the screen.

3.2. Booting in Response to Packets

If DMT receives a request-for-connection (RFC) Pup and DP0 is ready, then it boots the Operating system and passes it a message of type eventRFC. If the Executive section of user.cm contains an entry of the form:

eventRFC: <arbitrary command line>

then the executive will consume the event and execute the command line. << If DMT receives an EFTP data packet with sequence number 0 and DP0 is ready then it boots the OS and passes it a message of type eventEFTP. This is included so that printers (which use the EFTP protocol) can drop into DMT when nobody is using them, and automatically wake up when someone wants to print. >> If DMT receives a Kiss-of-Death Pup for socket 4 (miscellaneous services), then it EtherBoots the file whose ID is contained in the low 16 bits of the Pup ID.

4. Peek

Peek opens several windows on the display. The top window is for user commands. There is currently only one: Quit. The next window displays the release date of the program, a digital clock, the Pup internetwork address of the machine, and the number of free pages on the disk. The next window is opened by the Peek Server and displays DMT reports as they arrive.

Peek loads special Ethernet microcode so that it can receive Peek reports directed to host 376b as well as conduct business as itself. If it can't load the ram, it runs the Ether interface promiscuously and filter packets in software. More diagnostic reports will be lost and booting may be slower, but things should still work.

Peek has a lot of options, and reads User.cm to find out what to do. An example of the Peek slice of a User.cm file is given below. In addition, it contains a host of network servers:

4.1. Peek Server

If there is a line of the form "Peek <filename>" in User.cm, Peek will start up a Peeking process which will listen for raw Ether packets of type PeekReport and write them on <filename>. The filename should be 'Peek.reports' since PeekSum, described below, assumes this (I was just feeling general the day I wrote that code).

4.2. Event Report Server

Peek implements the Pup Event Report protocol. For each line of the form "ERP <number> <filename>" in User.cm, Peek will instantiate an event report process which will listen on socket <number> and write event reports on <filename>. The default address which the OS uses is Maxc, so I don't expect many people will use this, however it might be helpful for an Alto site that isn't connected to the Parc Internet.

4.3. Pup Echo Server

Peek contains a Pup Echo server running continuously in the background. PupTest and GateControl contain Echo users with which you can poke it.

4.4. Raw Ether Echo Server

Peek also contains a raw Ethernet Echo server. This is the echo protocol used by EDP and NEDP, the diagnostic programs for the Alto and Nova Ethernet interfaces.

4.5. Boot Server

Peek implements the protocols necessary to be an Alto boot file server. For each line of the form "Boot <number> <filename>" in User.cm, Peek will send <filename> when it receives a Mayday packet requesting bootfile <number>. If the file isn't on the disk, or if Peek discovers a neighboring Boot server with a later version, your Peek will aquire it. The more boot files you tell Peek to keep, the less space there is for Peek reports.

4.6. Name Server

PeekSum consults the file 'Pup-Network.Directory' to get the owner and location of Altos. Peek contains a name lookup server and in addition to answering lookup requests, keeps its copy of the directory current.

4.7. Time Server

Peek also has a time server. Alto time is based on Greenwich Mean Time, and local users must know their local time zone and the beginning and ending days of Daylight Savings Time to convert to local time. Time servers are the source of this information, so it is important that the time parameters in User.cm be correct. "Zone +8:00" means that the peek disk is 8 hours west of Greenwich -- in the USA Pacific Time zone. The standard User.cm contains this, so you must edit it if you live elsewhere. The Daylight Savings Time parameters are set by the line "DST 121,305", and only change when Congress messes with time. Keep an eye on your local CongressPerson.

4.8. User.cm Example

Below is an example of the Peek part of a User.cm file. In this example DMT statistics go to the file 'Peek.reports', Event reports addressed to socket 30 (swat error reports) go to the file 'Swat.ERP', and some maintenance-type boot files are available for diagnosing Altos. Notice that all characters between a semicolon and a carriage return are considered to be comments and ignored by Peek (this is not true for all programs that use User.cm).

```
[EXECUTIVE]
...executive stuff...

[PEEK]
; Syntax:
; Boot <boot file number> <filename>
```

```
; ERP <socket number> <filename>
; Peek <filename>
; Correction <seconds per day> (decimal) [positive makes clock go faster]
; DST <beginning day> <ending day> (decimal)
; Probe <hours> (decimal)
; Zone <sign><hours>:<minutes> (decimal, plus is west of Greenwich)

Peek Peek.reports      ; for PeekSum.run

ERP 30 Swat.erp        ; Swat Error reports

Zone +8:00             ; USA Pacific Time Zone
DST 121,305            ; DST begins on day 121 and ends on day 305

Boot 0 DMT.boot
Boot 5 CRTTest.Boot
Boot 6 MadTest.Boot
Boot 10 NetExec.boot
Boot 11 PupTest.boot
Boot 12 EtherWatch.Boot
Boot 13 KeyTest.boot
Boot 15 DiEx.Boot
Boot 17 EDP.Boot
Boot 20 BFSTest.Boot

[BRAVO]
...bravo stuff...
```

Peek writes the contents of User.cm into the Command window as it reads through the file. If the file has bad syntax, Peek will call Swat with a description of its complaint (e.g. "[ReadNumber] - number contains illegal characters" if it is expecting a number and reads something other than 0-7). Typing <ctl>-U will restore the user display. The last item in the Command window is what Peek is having trouble with.

The source code for most of the servers in Peek is borrowed from the gateway program, and so there are some more specialized commands which you can ignore and which default to reasonable actions. I mention them here for completeness. "Correction +20" means the Alto's clock looses 20 seconds per day, and the time server should correct by gaining 1 second at 20 equally spaced times during a day. "Probe 1" means attempt to locate newer versions of boot files and the network directory once an hour.

5. PeekSum

PeekSum reads the file "Peek.Reports" (the output of Peek) and constructs a summary of the errors reported by DMT (see above) for each Alto. PeekSum writes on the file 'PeekSummary.Tx' a tabulation of the error reports, together with the owner's name and the machine's location, retrieved (if possible) from the file "Pup-Network.Directory", which is maintained by Peek, as described above.

As Peek is started and stopped, it writes short messages to this effect on Peek.Reports; these messages are reproduced at the beginning of PeekSummary.Tx. The number of the local network is also written. If Peek.Reports contains multiple reports from a single Alto (which is usually the case), PeekSum will record the largest number of errors of each type, over all such reports.

PeekSum will complain and then gracefully stop execution if the files Peek.Reports or PeekSummary.Tx are unopenable for some reason. If Pup-Network.Directory is unopenable or absent, the ouput file PeekSummary.Tx will not include names and locations of Altos, but will contain error reports grouped by Alto host number.

To run PeekSum, just type:

>PeekSum

and the program will go about its business. When it has finished, PeekSummary.Tx should be printed on your local printer.

DPrint - Diablo Printer Program

This program types text files on a Diablo printer connected to the Alto. It is a vanilla program with very few features. Use Bravo if DPrint's facilities are inadequate.

The syntax of the command line is:

DPrint/switch parameter/switch ... filename filename ...

The only switch permitted on the word "DPrint" is "/P", which instructs DPrint to pause before the beginning of each page.

One or more parameters may optionally be specified:

n/W Sets the line width to be n characters. Lines longer than this will wrap around to the next line. The default is 75 characters.

n/L Sets the page length to be n lines. This determines the point at which printout will pause (if /P was invoked) and also controls the amount of paper spewed when a form-feed is encountered in the file. The default is 66 lines (11 inches) if /P is not in effect or 57 lines (9.5 inches) if it is.

n/M Sets the left margin to be n units of 1/10 inch from the hardware left margin of the printer. The default is zero.

Command line parameters without switches are assumed to be names of text files to be printed. If a file cannot be found or a parameter is otherwise incorrect, you will be prompted for the correct value.

When DPrint pauses, you may either type space to resume printout or "Q" to abort it and quit out of the program. DPrint will pause immediately if you strike any key while it is printing, and also if the printer becomes not ready.

EmPress

EmPress has several functions. Its primary use is to convert ordinary text files into Press format, and to send the converted files to a Press printing server. Options include the ability to produce a Press file without transmitting it, and to transmit Press files that have been previously produced. Additional features provide for merging several Press page images into a single Press file, and for personalizing individual copies of documents.

EmPress can distinguish Press files from text files, so it need not be told whether to convert. As a text file converter, EmPress is intended for formatting program listings and supports only simple formatting operations such as Tab and FormFeed. Bravo trailers are ignored.

Joe Maleson wrote the original program. David Boggs made the modifications that allowed transmission of files to printers. Rick Tiberi produced the current version, adding the Press file merger and copy personalization facilities, and curing many problems.

Standard Case:

To send one or more Press or text files to your default Press printer, using a default font to convert the text files, type:

empress file1 file2 file3 ...

and read no further. The more general command line to EmPress is:

EmPress[/<global switches>] [<parameters>/<switches>] inputFiles

The square brackets denote portions of the command line that are optional and may be omitted. EmPress will print up to 100 input files.

Each global switch has a default value which is used if the switch is not explicitly set. To set a switch to 'false' proceed it with a 'minus' sign; to set it to 'true' just mention the switch.

Switch	Default	Function
/T	true	[Transmit] will send the resulting press file to a printer.
/number	8	(text files only) tab width -- see below.
/H	true	(text files only) [Headings] will print a heading and page number on each page.
/D	false	(press files only) [Date] will add the machine-readable time stamp to Press files that need them and don't have them. This allows Press files created by old software to print correctly. If your Press file prints with improper line justification and character spacing, try this switch before giving up.

EmPress recognizes a number of optional parameters which can be set from the command line. Parameters set from the command line take precedence over defaults built into the program.

Parameter	Default	Function
string/O	Swatee	[Output] the name of the output file. EmPress uses Swatee unless told otherwise, since the output press file is usually sent to the printer and then discarded.
number/C	1	[Copies] the number of copies to print.
string/H	none	[HostName] the name of the printer. This takes precedence over the name following PRESS: in the [HardCopy] section of User.cm.
string/I	none	[Input] the name of an input text file to be formatted and saved or transmitted, or of an input Press file to be transmitted.
string	none	a string without any switches is assumed to be an input file.
	The remaining switches apply to text conversion only.	
number/T	8	[Tab] the width of a tab character in multiples of the width if a space character.
string/F	Gacha	[FontFace] the font to use. You must have 'Fonts. Widths' on your disk.
number/P	8	[PointSize] the point size of the font.

User.Cm Entries

The following is a sample User.Cm hardcopy section, configured to use the Menlo Press printing server as the preferred printer:

```
[HARDCOPY]
PREFERREDFORMAT: Press
EARS: Palo
PRESS: Menlo
PRINTEDBY: "$"
FONT: TIMESROMAN 10 MIR
```

The FONT entry specifies that TimesRoman10i (italic) should be used as a default font instead of Gacha8 (EmPress's default choice). The second, point size argument, and the third, face specification argument are optional. The face argument contains three letters specifying weight (M, B, or L), slope (R or I), and expansion (C, R, or E), respectively.

The PRINTEDBY field, if present, specifies the name to be used in the Name field on the break page. The current disk login name will replace the character $. EmPress chooses "$" as a default in the absence of a specification.

Program operation

When EmPress encounters a Press file in the input list, it transmits (or stores) any text file that it is currently converting, then transmits the Press file. A new break page will be printed for each Press file, containing that file's name. EmPress will override the "created by" field of a Press file with a name derived as described above. It will fill in blank file name and date fields with the obvious defaults. If copies are specified in the command line, EmPress will override the number of copies specified in the Press file with the command line value.

EmPress uses the file Swatee for temporary storage while converting text for transmission. If in so doing Swatee becomes nearly full, EmPress will suspend formatting, send what has accumulated so far, and then press on. This has two desirable consequences: 1) a very full disk will not run out of space and 2) some pipelining can take place since the printer can munch on the first chunk while EmPress empressifies another.

Press File Merging

EmPress will merge several one page Press files into a single one page Press file. This allows the outputs of Bravo, Sil, Draw, Markup, etc., to be merged without a separate pass through Markup. One additional text or Press file may also be submitted, and it will be printed following the one page merge result.

One invokes the merge feature through one additional global switch, and one additional local switch:

Additional Global Switch:

/m Merge. All subsequent input files that are not qualified by switches must be single-page Press files. They will be merged to form a single (cover) page in the Press file result, containing all their Press specifications. This switch also conditions Empress to expect the additional local switches, described just below and in the Personalization section.

Additional Local Switch:

/d Document. This switch may be used to identify an optional main document, when the merge option is used. The file may be a simple text file or a Press file. It will follow the one page merge result in each copy printed.

Personalization

This relatively specialized feature is provided to allow the personalization of individual copies of a document. Each copy of the document might contain, for instance, the name and address of the person for whom it is intended. Up to six lines of personalized information can be specified. This information will replace distinctive "key strings" that have been placed in the cover page (merged) files or in the main document.

The key strings must appear in contiguous groups of up to six lines each. The personalized information for the current copy, specified in a paragraph of a special Bravo-format addressee file or in the command line, will replace the key strings in each group, line for line. Thus the personalized information may occur more than once in each document (Dear Mr. PARC/SDD: ... yes, you and all the members of the PARC/SDD household can enjoy the benefits of ...). Lines in the addressee paragraph for which no keys are provided are discarded.

The default key is "<", forty hyphens ("-"), then ">". If the string "<--title-->" appears anywhere in the document, the name of the "main" document (the one specified using the "/d" switch) will replace it.

The "/m" (merge) global switch must be specified before any of these personalization specification switches are valid.

Additional Local Switches:

/k Key. The item is a key that replaces the default (see above).

/a Addressee. The item is either the name of a Bravo format file containing a list of addressees -- one per paragraph, one line in each paragraph for each key line in the cover page or main document -- or a literal addressee, enclosed in double quotes. In a literal, use hyphens where you wish blanks to appear in the name.

ERP - Event Report Protocol Server

ERP is an event report protocol server. You invoke it by saying to the Exec:

ERP <socket> <filename>

where <socket> is a 16-bit socket number (the high 16 bits are zero), and <filename> is the name of a file on your disk. It starts a Event report server on <socket> which appends events to <filename>. This program is merely a thin veneer on the PupERPServ package, whose documentation you should consult for the file format.

Executive User's Guide

Executive, the Alto command processing subsystem, is the intermediary by which Alto users generally invoke other subsystems and ask simple questions about the state of the Alto file system. It is just the same as any other subsystem, except that its name is known by the Alto Operating System, and it is invoked by the Operating System whenever the BCPL operator "finish" or equivalent is executed. This document describes version 10 of Executive.

1. What It Does

The operation of Executive proceeds thus:

1. It reads any leftover unexecuted commands from a file called Rem.Cm into a main memory command queue.

2. It begins building up a command line (terminated by a CR). If the command queue empties before the command line is terminated, additional characters are read from the keyboard until a CR is read. Editing is done during this phase. If the command line has been empty for about twenty minutes, the user is assumed to be occupied elsewhere, and the diagnostic program Dmt.Boot is invoked either from the disk (if it can be found) or from the Ethernet.

3. The edited command is placed at the front of the command queue and the command queue is analyzed for *-, #-, and @-substitutions. If something of the form @filename@ is discovered in the first line in the command queue, it is replaced by the contents of the named file and analysis continues with the first character of the replacement. Executive makes no attempt to detect or recover from infinitely recursive replacements. If the characters * or # are encountered in a filename in the first line, the file directory is used to replicate that filename with appropriate substitutions. This step results in a completely edited command line.

4. The first atom (contiguous sequence of legal file name characters) in the command line is analyzed to see whether it is the name of a subsystem in the file directory or the name of a command internal to Executive or neither. If neither, then Executive attempts to extend the atom into the name of a subsystem or Executive command. (The subsystem lookup algorithm is described below.) Failing in this, it complains and resets itself. Otherwise the line is written on the file Com.Cm. Then if the first atom was or could be extended into a subsystem name, the rest of the command queue is written on Rem.Cm, and the subsystem is invoked with a CallSubSys Operating System call. If it is an internal Executive command, the appropriate subroutine is called. The Executive passes the switches found on the subsystem name in the user parameters vector of CallSubSys. See the documentation of CallSubSys for more details.

In parallel with these steps, Executive does a few housekeeping chores:

a. It reads the entire file directory into memory, merges in the names of user-callable routines internal to Executive, and sorts the resulting list alphabetically.

b. Having nothing else to do, it puts a line containing a continuously-updating digital clock and the number of free disk pages on the user's screen, and flashes a cursor where the next typed character will go.

A number of characters have special meaning during the editing step (2):

Null:
Linefeed:
Ignored

Carriage Return:

Terminates the line, beginning step 3.

Control-A:
Backspace:
Removes the last character from the line queue.

Control-W:
Removes the last item which looks like a file name from the line queue.

UpArrow:
Single quote:
Causes itself and the next character both to be appended to the line queue, regardless of what the next character is.

Control-U:
Signals that at the conclusion of step 2 the line queue is to be written on the file Line.Cm and its contents replaced by the text "Bravo/n Line.Cm". If one has the proper Bravo and User.Cm, this will invoke Bravo on the command line. (This is also an easy way to build small command files. Just type the desired command followed by Control-U and CR. Then copy or rename Line.Cm.)

Control-X:
Performs step 3 on the line queue as it is, returns to step 2. In other words, it eXpands @, *, and #.

Control-C:
Delete:
Empties out the line queue, starts over again.

Escape:
Interprets the last atom in the line queue as the prefix of a file name; continues that file name until it is complete or ambiguous. Flashes the screen if it is ambiguous.

?:
Interprets the last atom of the line queue as the prefix of a file name; types out all file names which begin that way.

Tab:
Same as "?" except it deletes the atom from the line queue after typing the file names. This would be what one would normally use to interrogate the directory. * and # work as expected.

In step 3, several characters have special meaning:

Semicolon:
Carriage Return:
Terminate the line; control goes to step 4.

Up Arrow:
If followed by a carriage return, do nothing. If followed by an up arrow, put one up arrow in the line queue. If followed by any other character, put both characters in the line queue (Ugh!).

/:
If followed by another "/", this begins a comment, so scan ahead until finding a carriage return or semicolon. If not, put the "/" in the line queue.

@:
Scan ahead until finding another @ (the second @ may be omitted if it comes at the end of the command). The atom in between is a file name. Replace the @atom@ by the contents of the named file. If the file doesn't exist exactly as specified, try extending the specification and forcing a .Cm suffix.

*:
#:

Expand the atom using these characters by making a search through the file directory. * matches any sequence of file name characters. # matches any single character except a period. File names are defined to end with an infinite number of periods. The atom is replaced by all file names matching its pattern. Switches on the atom, if any, are replicated.

There is one special character recognized during step 4.

Control-C:

Aborts the command and starts over again. Control-C is effective up until the time that Executive gives up control to the subsystem being invoked. If you realize a mistake in your command after typing CR, quickly typing Control-C will abort it. (When Executive's header line disappears, it is too late.)

In step 4, one switch is taken to have special meaning on the subsystem name only. The switch /! will set the pause parameter in the call to CallSubSys to true allowing you to enter Swat after your program is loaded, but before its first instruction is executed. This switch, if detected, is removed from the command line before Com.Cm is created. This feature is extremely useful if your program is hitting a bug before its first user interaction.

2. Executive Commands

The Executive contains a number of subroutines which can be invoked from the command line. The commands corresponding to these subroutines can be identified by the extension character "~", which is illegal in a file name. Executive commands include the following:

Type.~ FileName ...

Display the contents of the named file(s) on the screen. After each page, it asks whether you want to see more of the current file. A Ctrl-C at this point terminates the entire Type command. You can type any files, even binary ones, but typing some files will give you more useful information than typing others.

Delete.~ FileName ...

Removes the named files from the directory and frees their disk space. Use this command very carefully. Its effect cannot be undone. Typing Ctrl-C will abort the command cleanly between deletions.

Copy.~ DestFileName ← SourceFileName ...

Copies a file. If there are several SourceFileNames then the copy will contain the concatenation of the information in the source files, in the order listed. In accordance with the Alto File Date Standard, copying a file preserves the creation date of the file; concatenating files generates a new creation date.

Rename.~ OldFileName NewFileName (or NewFileName ← OldFileName)

Changes the name of OldFileName. NewFileName must not already exist unless OldFileName and NewFileName are the same (use this feature to change the capitalization of a file name).

BootFrom.~ FileName [...Sys.Boot]

Initiates a software-simulated bootstrap sequence on the file named by FileName. Most probably the FileName should have the .Boot extension. Like the OS system call BootFrom (which it uses) this command does not actually do a hardware bootstrap operation, so it does not re-initialize any Alto hardware or microcode tasks. If you don't know what this implies, don't worry about it.

Quit.~

Diagnose.~

Has the effect of BootFrom Dmt.Boot. This commences the running of the diagnostic program, which doesn't use the Operating System at all. This is done automatically after a machine has been idle in Executive for about 20 minutes. If Dmt.Boot is not on your disk or you turn the disk off Dmt will be loaded from the Ethernet.

Login.~
Places your user name and password in the system area of main memory for use by programs which interact with access-controlled resources (like timesharing or file systems, for example).

SetTime.~
Sets the Alto's internal time-of-day clock. The time is obtained from the Ethernet if possible. Failing that you will be asked to supply the time (and possibly time zone) manually in the form 12-jan-78 14:45. Use SetTime/m to bypass the Ethernet and set time manually. Use /z to force setting of time zone in manual mode. (When Executive is started it examines the time-of-day clock. If the value is not reasonable Executive attempts to obtain the time from the Ethernet before proceeding. If the time cannot be obtained, the time-of-day displayed at the top of the screen will be "Date and Time Unknown" indicating that you should invoke the SetTime.~ command manually.)

Dump.~ DumpFileName SourceFileName ...
Writes DumpFile as a structured file (in Dump format) containing the names and data of all the SourceFiles. This is a convenient way of packaging up a collection of related files into a single composite file that can later be decomposed into its constituent parts. See Appendix A for details of Dump format. The primary virtue of this particular format is that it is intended to be compatible with the Dump format of the Data General Nova DOS operating system, and it is compatible with the Tenex subsystem DUMP-LOAD.SAV.

Load.~ DumpFileName
This reads through a Dump format file and creates individual files corresponding to its constituent parts. The /V switch causes Load to ask you about each constituent part, whether to copy it from the DumpFile to an individual file or not. Acceptable responses are Y, N, and C. The latter indicates that you would like it to be copied, but into a file with a different name than that indicated. You are then asked to supply the name you prefer.

Release.~
Tells you the release number and date of Executive. The release number is also shown in the first Executive herald line, just after the slash following "Xerox Alto Executive."

StandardRam.~
For any Trap except the Swat Trap (#77400) the Alto microcode sends control of the emulator task to the microcode Ram for interpretation. StandardRam initializes the microcode Ram to send control of the emulator task back to the Rom Trap-handling microcode. If you don't initialize the microcode Ram before executing a program which 1) uses Traps, and 2) doesn't initialize the Ram itself, then when the first Trap happens your machine will probably do something bizarre, but it usually will not destroy disk data.

Install.~ FileName [...Sys.Boot]
Causes a customized version of the operating system on the file named by FileName to be written on the file Sys.Boot. For further details, please see the section on "Installing the operating system" in the Alto Operating System manual.

BootKeys.~ FileName [...Sys.Boot]
Did you know that by holding down various combinations of keys on the Alto keyboard while pressing the boot button it is possible to get the Alto to bootstrap load itself from any file on the disk? (This bootstrap will probably crash fairly quickly on any file except one in .Boot format.) Bootstrapping the Operating system is simply a special case of this: all keyboard keys up refers to disk address 0, which by convention is where a copy of the first data page of Sys.Boot is stored. To find out what keys to push in order to bootstrap load other files, you use the BootKeys command.

Resume.~ FileName [...Swatee]
The file named by FileName is patched so that its Swatee file pointer is the same as the current Swatee file pointer, and then it is loaded in and run. For best results, this file should be Swatee, or a copy of a Swatee. If you want to return to Swat with an old Swatee (for example, originally you didn't have the right .SYMS file) you can say

```
Copy.~ Swatee ← OldSwatee (no need to do this if Swatee is already correct)
Resume.~ Swat
```

Chat.~
Ftp.~
Scavenger.~
NetExec.~
These commands load the corresponding programs from the Ethernet. If you have the .Run file for one of these, it will be found instead by the normal Executive lookup strategy.

EtherBoot.~ octal number
This command will boot any available Ethernet bootable file provided that you know its number.

FileStat.~ FileName ...
This command will tell you several things about a file: its length in bytes, size in pages, serial number and disk address, creation and write dates. If any FileName is of the form octal/s (or octal1,octal2/s) the file will be looked up by serial number rather than by name. This is useful if Scavenger or some other program gives you a serial number without telling you the name. The forms octal/v and octal/r tell you about the file that owns the specified virtual or real disk address.

3. Subsystem Lookup

Executive recognizes and knows how to invoke several kinds of subsystems. In order to select a subsystem matching the name given in the command line Executive uses the following algorithm:

1. For each of the strings <null>, ".run", ".image", ".bcd", ".~", "*.run", "*.image", "*.~" and "*.bcd" ask how many directory entries are matched by appending the string to the typed name. As soon as the answer is one the subsystem is found. Note that the question is asked separately for each extending string and that the questions are asked in the order specified. The order of the search means that the order of subsystem types is: Bcpl program, Mesa image file, Mesa bcd file, internal command (the order of Mesa bcd files and internal commands is reversed if the name is not completely specified).

2. If the subsystem name ends in ".image" it is assumed to be a Mesa image file and is invoked using the program RunMesa.run.

3. If the subsystem name ends in ".bcd" it is assumed to be a runnable Mesa configuration. "Mesa.image" is added to the front of the command and the lookup starts over.

4. Otherwise the subsystem is invoked directly (if internal) or via CallSubsys.

4. User.Cm Entries

The Executive section of User.Cm may contain several commands to the Executive. Most of these are command lines to be executed if some event is noted (see the Operating System documentation for a description of events). In addition to standard events, any other event may be specified using the notation eventN where N is the event number (in decimal).

The number of text lines in the user command window can be set from User.Cm using the selector DisplayLines: followed by a number. You are advised not to set this number higher than its default value (currently 16), but you might want to reduce the number in order to leave more memory space for your directory if you have a large number of files (say, more than 400).

The line "Screen: Black" in User.Cm directs Executive to use the display in white-on-black rather than the normal black-on-white mode.

5. Dump Format

A dump file is a sequence of blocks of eight-bit bytes. The first byte of each block is the block type. A typical dump file might look like:

<name block><data block 1>...<data block n>
.
.
<name block><data block 1>...<data block m>
<end block>

Name Block - Type = #377

A name block contains two bytes of file attributes and then the file name. The file attributes are used by the Nova operating system; Alto Dump.~ sets these bytes to 0, and Alto Load.~ ignores them. The file name is a sequence of ASCII characters terminated by a 0 byte.

Data Block - Type = #376

A data block contains two bytes of byte count (high-order byte first), two bytes of checksum (high-order byte first), and a sequence of data bytes. The byte count must be less than or equal to 256 for compatibility with Novas, and the count does not include the checksum or byte count; only the data bytes are counted. Novas do not handle data blocks with byte counts of 0 or 1 correctly. Alto Dump.~ will not produce such blocks unless forced to dump a file whose length is less than 2 bytes. The checksum is a 16-bit add ignoring carry, over the data and byte count. If the block has an odd number of bytes, the last byte is NOT included in the checksum computation.

Error Block - Type = #375

Novas generate error blocks. Alto Dump.~ does not. Alto Load.~ terminates if it encounters one.

End Block - Type = #374

An end block has no contents and terminates a Load.~.

Date Block - Type = #373

Date blocks with six bytes of date are generated by Nova RDOS. Alto Dump.~ does not generate them; Alto Load.~ ignores them.

N.B. This appendix is included thanks to David Boggs.

Find - a file searching subsystem

The Find subsystem allows you to search text files at very high speed on an Alto. Examples of such files might be an address/telephone list, a source program, or a library catalog.

Find basically looks for all the occurrences of a pattern in a file, just like doing repeated Jump commands in Bravo. A pattern is just a character sequence, except for the following:

in a pattern means "any character at all", e.g. CAP and CUP count as occurrences of the pattern C#P.

~ in a pattern means "allow one character in the occurrence to disagree with the corresponding character in the pattern". For example, LAP, CUP, and CAT all count as occurrences of the pattern ~CAP (or CAP~ or C~AP). Two ~s mean "allow two disagreements", and so on. Note that "disagreement" only means substituting one character for another: it does not include insertions (e.g. CLAP for CAP), deletions (CP for CAP), or transpositions (CPA for CAP).

If you really want to have a pattern containing # or ~, you have to type a ' before it, e.g. to search for the character sequence ATOM #, you have to type ATOM '#.

Unless you use the /c switch described below, upper and lower case letters are considered identical, e.g. Cap, cap, and CAP all count as occurrences of CAP or of cap.

Unless you use the /s switch described below, blanks (spaces) in the file are completely ignored, e.g. C A P counts as an occurrence of CAP; blanks in the pattern are also ignored.

There are two ways to invoke Find. The first way just searches a file for one pattern:

>Find filename pattern

(Since the Executive does something special about @, #, %, *, ↑, and ; in command lines, you must precede any of these characters in your pattern by a '. This is in addition to any 's you may need as described in the preceding paragraph.) The second way only specifies the file:

>Find filename

and Find then prompts you repeatedly for patterns. To leave Find when using it this way, use shift-Swat or type an empty pattern (just type <return> when Find says Pattern:). You can also use Find to search several files together, by invoking it with

>Find/m filename1 ... filenamen

which also prompts you for patterns.

In any of the above command lines, you can also use the /s switch, i.e. any of the forms

>Find/s filename pattern
>Find/s filename
>Find/ms filename1 ... filenamen

This causes Find to treat spaces (blanks) just the same as any other character in the file and in the pattern.

In any of the above command lines, you can also use the /c switch, which causes Find to treat upper and lower case letters as different from each other.

After completing the search, Find displays at the top of the screen a summary of the form:

79 occurrences, 1200 ms, 150 pages

giving the total number of occurrences, the time in milliseconds, and the number of disk pages in the file. In the remainder of the screen, Find displays the line containing each occurrence of a pattern, with the occurrence indicated in boldface. To the left of the line, Find displays the character position in the file where the occurrence was found. After each screenful, Find waits for you to type <space> if you want more, or <del> if you don't.

In addition to displaying matches on the screen, Find always writes the lines containing matches on a file called Find.Matches. However, it only writes those matches which it displayed, so if you stopped the display of matches with <del>, only those matches you actually saw will appear on the file.

What Find finds for you is all the "items" containing occurrences of the pattern. Normally an "item" is just a single line of text, delimited by <cr> on both ends. However, Find also knows about two other kinds of items: Bravo paragraphs, and groups of lines separated from each other by a blank line. If you use the

/p (for Paragraph) switch, Find will show (display and write on Find.Matches) the entire Bravo paragraph containing the occurrence. If you use the /b (for Blank line) switch, Find will show everything surrounding the occurrence up to the next preceding and following blank line.

Find produces a large number of error messages. The messages

Pattern too long
Can't preallocate
RAM full

all mean the same thing, namely that your pattern is too long or too complicated (unfortunately, it is too complicated to explain exactly what "too complicated" means). The message

Can't load RAM

means that your Alto has old or non-standard ROMs and Find can't do what it needs to do: you should contact a hardware maintainer. (This should never happen on Alto II's.)

Find has many obvious limitations. They can all be removed if people complain about them. The following features could also be added upon request:

Requiring that a match be delimited by non-alphanumerics.
Boolean combinations of matches, maybe.
Ability to work with Trident disks.
Possibly other features requested by users.

Programmers should note that the file searching capability is also available as a library package (called FindPkg), so programs can use it as well as people.

History of changes:

Release of 1/16/78

Added /c (distinguish upper and lower case), /p (item = paragraph), and /b (item = between blank lines) switches.

Alto Pup File Transfer Program

FTP is a Pup-based File Transfer Program for moving files to and from an Alto file system. The program comes in 3 parts:

1) An FTP Server, which listens for file transfer requests from other hosts,

2) An FTP User, which initiates file transfers under control of either the keyboard or the command line, and

3) A User Telnet for logging into a remote host using the Pup Telnet protocol.

1. Concepts and Terminology

Tranferring a file from one machine (or "host") to another over a network requires the active cooperation of programs on both machines. In a typical scenario for file transfer, a human user (or a program acting on his behalf) invokes a program called an "FTP User" and directs it to establish contact with an "FTP Server" program on another machine. Once contact has been established, the FTP User initiates requests and supplies parameters for the actual transfer of files, which the User and Server proceed to carry out cooperatively. The FTP User and FTP Server roles differ in that the FTP User interacts with the human user (usually through some sort of keyboard interpreter) and takes the initiative in user/server interactions, whereas the FTP Server plays a comparatively passive role.

The question of which machine is the FTP User and which is the FTP Server is completely independent of the direction of file transfer. The two basic file transfer operations are called "Retrieve" and "Store"; the Retrieve operation causes a file to move from Server to User, whereas Store causes a file to move from User to Server.

The Alto FTP subsystem contains both an FTP User and an FTP Server, running as independent processes. Therefore, to transfer files between a pair of Altos, one should start up the FTP subsystem on both machines, then issue commands to the FTP User process on one machine directing it to establish contact with the FTP Server process in the other machine. Subsequent file transfers are controlled entirely from the FTP User end, with no human intervention required at the Server machine.

Transferring files to or from a Maxc system or an IFS involves establishing contact with FTP Server processes that run all the time on those machines. Hence, one may simply invoke the Alto FTP subsystem and direct its FTP User process to connect to the machine.

In the descriptions that follow, the terms "local" and "remote" are relative to the machine on which the FTP User program is active. That is, we speak of typing commands to our "local" FTP User program and directing it to establish contact with an FTP Server on some "remote" machine. A Retrieve command then copies a file from the "remote" file system to the "local" file system, whereas a Store command copies a file from the "local" file system to the "remote" file system.

Furthermore, we refer to "local" and "remote" filenames. These must conform to the conventions used by the "local" and "remote" host computers, which may be dissimilar (for example, Alto versus Maxc). The Alto FTP knows nothing about Maxc filename conventions or vice versa.

The Alto FTP subsystem also includes a third process, called a "User Telnet", which simulates a terminal in a manner exactly analogous to the Chat subsystem (though lacking some of its finer features). By this means, you may log in to a file sytem machine to perform operations not directly available via the basic file transfer mechanisms. If you log into Maxc, it is even possible to run "PupFTP", the Maxc FTP User program, and direct it to establish contact with the FTP Server in your own Alto. You should probably not

try this unless you really understand what you are doing, however, since the terms "local" and "remote" are relative to Maxc rather than to your Alto (since the FTP User program is running on Maxc in this case), which can be confusing.

2. Calling the FTP Subsystem

A number of options are available when running FTP. The program decides which parts of itself to enable and where user commands will come from by inspecting the command line. The general form of the command line to invoke FTP looks like:

FTP[/<Global-switches>] [<Host-name> [<Command-list>]]

The square brackets denote portions of the command line that are optional and may be omitted.

Global switches, explained below, select some global program options such as using the Trident disk instead of the Diablo. The first token after the <global-switches>, if present, is assumed to be a <host-name> (a discussion of which appears later in the description of the "Open" command). The User FTP will attempt to connect to the FTP Server on that host. After connecting to the server, if a <command-list> is present, an interpreter is started which feeds these commands to the User FTP. When the command list is exhausted, FTP returns to the Alto Executive. If no command list is present, an interactive keyboard command interpreter is started.

Each global switch has a default value which is used if the switch is not explicitly set. To set a switch to 'false' proceed it with a 'minus' sign (thus FTP/-S means 'no Server'), to set a switch to 'true' just mention the switch.

Switch	Default	Function
/S	true	[Server] starts the FTP Server. The Server is not started if the User is enabled and is being controlled from the command line.
/U	true	[User] starts the FTP User. As explained above, the interactive command interpreter or the command line interpreter will be started depending on the contents of the command line.
/C	true	[Chat] starts the Telnet. The Telnet is not started if the User is enabled and is being controlled from the command line, or if the system disk is a Trident.
/T	false	[Trident] sets the system disk to be a Trident drive. The default is 0, but can be changed by following the /T with a unit number. The unit number is octal; the high byte is the logical filesystem number and the low byte is the physical drive number. User and Server commands apply to files on this disk but command line input is still taken from Com.cm on the Diablo drive.
/L	*	[Log] causes all output to the User FTP window to also go to the file "FTP.log" on DP0, overwriting the previous contents. Log is true if the User is enabled and is being controlled from the command line.
/A	false	[AppendLog] enables the log but appends to FTP.log rather than overwriting it.
/E	true	[Error] causes FTP to ask you if you want to continue when a non-fatal error happens during execution of a command line. FTP/-E will cause FTP to recover automatically from non fatal errors without consulting you.
/R	true	[Ram] allows FTP to use some microcode which speeds things up slightly. If your Alto has no ram, this switch is ignored.

/D	false	[Debug] starts FTP in debug mode.

The rest of the global switches are explained below under 'Server Options'.

2.1. FTP User Log

FTP can keep a log (typescript) file for the FTP User window. The file name is 'FTP.log'. It is always enabled when FTP is being controlled from the command line; otherwise it is controlled by the /L and /A global switches. Some keyboard commands do not treat the user window as a simple teletype, so the typescript for these commands will not be exactly what you saw, sigh.

2.2. Using a Trident Disk

Starting FTP with the /T global switch causes FTP to store and retreive files from a Trident disk. Accessing a file on a Trident requires more code and more free storage than accessing a file on the Diablo. Since FTP is very short on space, only a User or a Server FTP is started when the /T switch is set. The default is to start a User FTP, but specifying no user (FTP/T-U) or specifying a server (FTP/TS) will start a Server FTP instead.

2.3. Server Options

Server options are controlled by switches on the subsystem name and subcommands of the SERVER keyboard command. There are currently four options:

switch	Default	Function
none		If no server option is specified, retrieve requests (disk to net) are allowed. Store requests (net to disk) are allowed unless the store would overwrite an already existing file. Delete and Rename are not permitted.
/P	false	[Protected] Retrieve requests are allowed. No stores are allowed. Delete and Rename are not permitted.
/O	false	[Overwrite] Retrieve requests are allowed. Store requests can overwrite files. Delete and rename are permitted.
/K	false	[Kill] FTP will return to the Alto Exec when the server connection is closed. A simple form of remote job entry can be performed if the user FTP stores into Rem.cm (Com.cm on Novas).

3. The FTP Display

The top inch or so of the display contains a title line and an error window. The title line displays the release date of that version of FTP, the current date and time, the machine's internetwork address, and the number of free pages on the disk. The error window displays certain error messages if they arrive from the network (errors are discussed in more detail below). A window is created below the title line for each part of FTP which is enabled during a session (server, user, and telnet).

If the FTP Server is enabled, it opens a window and identifies itself. If a User FTP subsequently connects to this Server, the User's network address will be displayed. The Server will log the commands it carries out on behalf of the remote User in this window. The Server is not enabled when FTP is being controlled from the command line.

The FTP User opens the next window down and identifies itself. The command herald is an asterisk.

The User Telnet opens the bottommost window, identifies itself, and waits for a host name to be entered. The Telnet is not enabled when FTP is being controlled from the command line.

4. Keyboard Command Syntax

FTP's interactive command interpreter presents a user interface very similar to that of the Alto Executive. Its command structure is also very similar to that of the Maxc Pup FTP program (PupFTP), and the Maxc ArpaNet FTP program (FTP). The standard editing characters, command recognition features, and help facility (via "?") are available. When FTP is waiting for keyboard input, a blinking cursor appears at the next character position.

4.1. Directing Keyboard input to the User and Telnet Windows

The bottom two unmarked keys control which window gets characters from the keyboard. Hitting the unmarked key to the right of 'right-shift' (also known as the 'Swat key') directs keyboard input to the Telnet window. Hitting the unmarked key to the right of the 'return' key (also known as the 'Chat key') directs keyboard input to the FTP User window. The window which currently owns the keyboard will blink a cursor at the next character position if it is waiting for type-in.

4.2. Keyboard Commands

OPEN <host name>

Opens a connection to the FTP Server in the specified host. FTP permits only one user connection at a time. In most cases the word OPEN may be omitted: i.e., a well formed <host name> is a legal command and implies a request to OPEN a connection. FTP will try for one minute to connect to the specified host. If you made a mistake typing the host name and wish to abort the connection attempt, hit the middle unmarked key (to the right of <return>).

Ordinarily, host name should be the name of the machine you wish to connect to (e.g., "Maxc"). Most Altos and Novas have names which are registered in Name Lookup Servers. So long as a name lookup server is available, FTP is able to obtain the information necessary to translate a known host name to an inter-network address.

If the host name of the server machine is not known or if no name lookup servers are available, you may specify an inter-network address in place of the host name. The general form of an inter-network address is:

<network> # <host> # <socket>

where each of the three fields is an octal number. The <network> number designates the network to which the Server host is connected (which may be different from the one to which the User host is connected); this (along with the "#" that follows it) may be omitted if the Server and User are known to be connected to the same network. The <host> number designates the Server host's address on that network. The <socket> number designates the actual Server process on that host; ordinarily it should be omitted, since the default is the regular FTP server socket. Hence, to connect to the FTP server running in Alto host number 123 on the directly-connected Ethernet, you should say "OPEN 123#" (the trailing "#" is required).

CLOSE

Closes the currently open User FTP connection. CLOSE cancels any defaults set by CONNECT, DIRECTORY, DEVICE, BYTE, TYPE, or EOLC commands.

LOGIN <user name> <password>

Supplies any login parameters required by the remote server before it will permit file transfers. FTP will use the user name and password in the Operating System, if they are there. Logging into FTP will set the user name and password in the OS (in the same manner as the Alto Executive's "Login" command).

When you issue the "Login" command, FTP will first display the existing user name known to the OS. If you now type a space, FTP will prompt you for a password, whereas if you want to provide a different user name, you should first type that name (which will replace the previous one) followed by a space. The command may be terminated by carriage return after entering the user name to omit entering the password.

The parameters are not immediately checked for legality, but rather are sent to the server for checking when the next file transfer command is issued. If a command is refused by the server because the name or password is incorrect, FTP will prompt you as if you had issued the LOGIN command and then retry the transfer request. Hitting delete in this context will abort the command.

A user name and password must be supplied when transferring files to and from a Maxc system or an IFS. The Alto FTP Server requires a user-password to be supplied if the server machine's disk is password-protected and if the password in the server machine's OS does not match the password on the disk. Thus if the OS was booted and FTP invoked because a Request-for-Connection was received (which bypasses password checking), FTP will refuse access to files unless a password is supplied. However if the OS was booted normally, FTP assumes that the disk owner (who knew the password) will control access by using the server option switches. The user-name is ignored.

CONNECT <directory name> <password>

Requests the FTP server to "connect" you to the specified directory on the remote system, i.e., to give you owner-like access to it. The password may be omitted by typing carriage return after the directory name. As with LOGIN, these parameters are not verified until the next transfer command is issued. CONNECT cancels the effect of any previous DIRECTORY command. At present, the "Connect" command is meaningful only when transferring files to or from a Maxc system or an IFS; the Alto FTP server currently ignores connect requests. If the "multiple directory" feature of the Alto Operating System ever comes into widespread use, this may be changed.

DIRECTORY <directory name>

Causes <directory name> to be used as the default remote directory in data transfer commands (essentially it causes <directory-name> to be attached to all remote filenames that do not explicitly mention a directory). Specifying a default directory in no way modifies your access privileges, whereas CONNECTing gives you 'owner access' (and usually requires a password). Explicitly mentioning a directory in a file name overrides the default directory, which overrides the connected directory, which overrides the login directory. Punctuation separating <directory name> from other parts of a remote filename should not be included. For example you might type "Directory Alto" not "Directory <Alto>".

RETRIEVE <remote filename>

Initiates transfer of the specified remote file to the local host. The syntax of <remote filename> must conform to the remote host's file system name conventions. Before transferring a file, FTP will suggest a local-filename (generally the same as the remote-filename without directory or version), and will tell you whether or not the file already exists on your local disk. At this point you may make one of three choices:

1. Type Carriage Return to cause the data to be transferred to the local filename.
2. Type Delete to indicate that the file is not to be transferred.
3. Type any desired local filename followed by Return. The previous local filename will disappear, the new filename will replace it, and FTP will tell you whether a file exists with that name. This filename must conform to local conventions. You now have the same three choices.

If the remote-filename designates multiple files (the remote host permits "*" or some equivalent in file names), each file will be transferred separately and FTP will ask you to make one of the above three choies for each file. At present, only Maxc and IFS support this capability. That is, you may supply "*"s in the remote-filename when retrieving files from a Maxc or an IFS, but not when retrieving files from another Alto.

STORE <local filename>
Initiates transfer of the specified local file to the remote host. Alto file name conventions apply to the <local filename>; "*" expansion is not supported. FTP will suggest a remote-filename to which you should respond in a manner similar to that described under RETRIEVE except that if you supply a different filename, it must conform to the remote file system's conventions. The default remote filename is one with the same name and extension as the local file; the remote server defaults other fields as necessary. If the remote host is a Maxc system or an IFS, then the directory is that most recently supplied in LOGIN or CONNECT or DIRECTORY commands and the version is the next higher.

DUMP <remote filename>
Bundles together a group of files from the local file system into a 'dump-format' file (see the Alto Executive documentation for the dump-file format and more on dump-files in general) and stores the result as <remote filename>. FTP will ask you for the names of local files to include in the dump-file. Terminate the dump by typing just <return> when FTP asks for another filename. By convention, files in dump-format have extension '.dm'.

LOAD <remote filename>
Performs the inverse operation of DUMP, unbundling a dump-format file from the remote file system and storing the constituent files in the local file system. For each file in the dump-file, FTP will suggest a local file name and tell you whether a file by that name exists on your disk. You should respond in the manner described under RETRIEVE.

LIST <remote file designator>
Lists all files in the remote file system which correspond to <remote file designator>. The remote file designator must conform to file naming conventions on the remote host, and may designate multiple files if "*" expansion or some equivalent is supported there. If the <remote file designator> is terminated by a comma rather than a carriage-return, FTP prints a prompt of "**" at the left margin and prepares to accept one or more subcommands. These subcommands request printout of additional information about each file. To terminate subcommand input, type a <return> in response to the subcommand prompt. The subcommands are:

Type	Print file type and byte size.
Length	Print length of file in bytes.
Creation	Print date of creation.
Write	Print date of last write.
Read	Print date of last read.
Times	Print times as well as dates.
Author	Print author (creator) of file.
Verbose	Same as Type+Write+Read+Author.
Everything	Print all information about the file.

This information is only as reliable as the Server that provided it, and not all Servers provide all of these file properties. Altos derive much of this information from hints, so do not be alarmed if it is sometimes wrong.

DELETE <remote filename>
Deletes <remote filename> from the remote filesystem. The syntax of the remote filename must conform to the remote host's file system name conventions. After determining that the remote file exists, FTP asks you to confirm your intention to delete it. If the remote filename designates multiple files (the remote host permits "*" or some equivalent in file names), FTP asks you to confirm the deletion of each file.

RENAME <old filename> <new filename>

Renames <old filename> in the remote filesystem to be <new filename>. The syntax of the two filenames must conform to the remote host's file system name conventions, and each filename must specify exactly one file.

QUIT
Returns control to the Alto Executive, closing all open connections.

TYPE <data type>
Forces the data to be interpreted according to the specified <data type>, which may be TEXT, BINARY. Initially the type is UNSPECIFIED, meaning that the source process should, if possible, decide on the appropriate type based on local information.

BYTE-SIZE <decimal number>
Applicable only to files of type Binary, BYTE-SIZE specifies the logical byte size of the data to be transferred. The default is 8.

EOL <convention>
Applicable only to files of type Text, EOL specifies the End-of-Line Convention to be used for transferring text files. The values for <convention> are CR, CRLF, and TRANSPARENT. The default is CR.

DEVICE <device>
Causes <device> to be used as the default device in data transfer commands (essentially it causes <device> to be attached to all remote filenames that do not explicitly mention one). The punctuation separating <device> from the other components of a remote filename should not be included. For example you might specify "Device DSK" to Tenex, not "Device DSK:"

USER
Allows you to toggle switches which control operation of the FTP User. There is currently only one: DEBUG, which controls display of protocol interactions. Warning: this printout (and the corresponding one in the SERVER command below) sometimes includes passwords.

SERVER
Allows you to toggle switches which control operation of the FTP Server. The switches are PROTECTED, OVERWRITE, KILL, and DEBUG, corresponding to the global switches /P, /O, /K, and /D.

TELNET
Allows you to toggle switches which control operation of the Telnet. There is currently only one: CLOSE, which closes the Telnet connection if one is open, and clears the Telnet window.

5. Command Line Syntax

The User FTP can also be controlled from the command line. As explained above, the first token after the subsystem name and server switches must be a legal host name; if the User FTP can't connect to the FTP Server on that host it will abort and return control to the Alto Executive. If a command list follows the host name, the command line interpreter is invoked instead of the interactive keyboard interpreter. This permits the full capabilities of the Alto Executive (filename recognition, "*" expansion, command files, etc.) to be used in constructing commands for FTP.

Each command is of the form:

<Keyword>/<SwitchList> <arg> ... <arg>

To get a special character (any one of "*#';") past the Alto Executive, it must be preceded by a single quote. To get a "/" into an FTP argument, the "/" must be proceeded by two single quotes (the second

one tells FTP to treat the "/" as an ordinary character in the argument, and the first one gets the second one past the Alto Executive).

Unambiguous abbreviations of command keywords (which in most cases amount to the first letter) are legal. However, when constructing command files, you should always spell commands in full, since the uniqueness of abbreviations in the present version of FTP is not guaranteed in future versions.

A command is distinguished from arguments to the previous command by having a switch on it, so every command must have at least one switch. The switch "/C" has no special meaning and should be used on commands where no other switches are needed or desired.

5.1. Command Line Errors

Command line errors fall into three groups: syntax errors, file errors, and connection errors. FTP can recover from some of these, though it leaves the decision about whether to try up to you.

Syntax errors such as unrecognized commands or the wrong number of arguments to a command cause FTP's command interpreter to get out of sync with the command file. FTP can recover from syntax errors by simply ignoring text until it encounters another command (i.e. another token with a switch).

File errors such as trying to retrieve a file which does not exist are relatively harmless. FTP recovers from file errors by skipping the offending file.

Connection errors such as executing a store command when there is no open connection could cause FTP to crash. FTP can't recover from connection errors.

When FTP detects an error, it displays an error message in the User window. If the error is fatal, FTP waits for you to type any character and then aborts, causing the Alto Executive to flush the rest of the command line, including any commands to invoke other subsytems after FTP. If FTP can recover from the error, it asks you to confirm whether you wish to continue. If you confirm, it plunges on, otherwise it aborts. The confirmation request can be bypassed by invoking FTP with the global error switch false (FTP/-E ...) in which case it will plunge on after all non fatal errors. If you aren't around when an error happens and you have told FTP to get confirmation before continuing after an error, the remote Server will probably time out and close the connection. If you then return and tell FTP to continue, it will get a fatal connection error and abort.

5.2. Command Line Commands

OPEN/C <host name>
: See description in "Keyboard commands". The first token after the subsystem name and global switches is assumed to be a host name and no OPEN verb is required (in fact if you supply it, FTP will try to make a connection the host named OPEN which is almost certainly not what you want).

CLOSE/C
: Closes the currently open User FTP connection.

LOGIN/C <user name> <password>
: See description in "Keyboard commands". The <password> may be omitted.

LOGIN/Q <user name>
: Causes FTP to prompt you for the password. This form of LOGIN should be used in command files since including passwords in command files is a bad practice.

CONNECT/C <directory name> <password>
: See description in "Keyboard commands". The <password> may be omitted.

CONNECT/Q <directory name>

Causes FTP to prompt you for the password needed to connect to the specified <directory name>. This form of CONNECT should be used in command files since including passwords in command files is a bad practice.

DIRECTORY/C <default directory>
See discription in "Keyboard commands".

RETRIEVE/C <remote filename> ... <remote filename>
Retrieves each <remote filename>, constructing a local file name from the actual remote file name as received from the Server. FTP will overwrite an existing file unless the /N (No overwrite) switch is appended to the RETRIEVE command keyword.

If the remote host allows "*" (or some equivalent) in a filename, a single remote filename may result in the retrieval of several files. (Note that you must quote the "*" to get it past the Alto Executive's command scanner.) As mentioned previously, this capability is implemented only by Maxc and IFS FTP Servers at present.

RETRIEVE/S <remote filename> <local filename>
Retrieves <remote filename> and names it <local filename> in the local file system. This version of RETRIEVE must have exactly two arguments. FTP will overwrite an existing file unless the /N (No overwrite) switch is also appended to the RETRIEVE command keyword. The remote filename should not cause the server to send multiple files.

RETRIEVE/U <remote filename> ... <remote filename>
Retrieves <remote filename> if its creation date is later than the creation date of the local file. A file will not be retrieved unless a local file with name and extension equal to the name and extension of the remote filename exists, or if the FTP server does not send a CREATION-DATE property. This option can be combined with RETRIEVE/S to rename the file as it is transferred.

RETRIEVE/V
Requests confirmation from the keyboard before writing a local file. This option is useful in combination with the Update option since creation date is not a fool-proof criterion for updating a file.

STORE/C <local filename> ... <local filename>
Stores each <local filename> on the remote host, constructing a remote filename from the name body of the local filename. A local filename may contain "*", since it will be expanded by the Alto Executive into the actual list of filenames before the FTP subsystem is invoked.

STORE/S <local filename> <remote filename>
Stores <local filename> on the remote host as <remote filename>. The remote filename must conform to the file name conventions of the remote host. This version of store must have exactly two arguments.

DUMP/C <remote filename> <local filename>...<local filename>
See the description in "keyboard Commands".

LOAD/C <remote filename>
See the description in "keyboard Commands". If the /V switch is appended to the LOAD command keyword, FTP will request confimation before writing each file. Type <return> to write the file, <del> to skip it. FTP will overwrite an existing file unless the /N (No overwrite) switch is appended to the LOAD command keyword.

DELETE/C <remote filename>
See the description in "Keyboard Commands". If the /V switch is appended to the DELETE command keyword, FTP will request confirmation before deleting each file. Type <return> to delete the file, and <del> (oops!) if you don't want to delete it.

COMPARE/C <remote filename>...<remote filename>

Compares the contents of <remote filename> with the file by the same name in the local file system. It tells you how long the files are if they are identical or the byte position of the first mismatch if they are not. (No corresponding command is available in the Keyboard command interpreter for implementation reasons: there is not enough room for it in Alto memory.)

COMPARE/S <remote filename> <local filename>
Compares <remote filename> with <local filename>. The remote filename must conform to the file name conventions of the remote host. This version of COMPARE must have exactly two arguments.

RENAME/C <old filename> <new filename>
See the description in "Keyboard Commands".

TYPE/C <data type>
See the description in "Keyboard Commands".

BYTE-SIZE/C <decimal number>
See the description in "Keyboard Commands".

EOL/C <convention>
See the description in "Keyboard Commands".

DEVICE/C
See the description in "Keyboard Commands".

DEBUG/C
See the description of the DEBUG subcommand under the USER command in "Keyboard Commands".

5.3. CLI Examples

To transfer files FTP.run and FTP.syms from the Alto called "Michelson" to the Alto called "Morley", one might start up FTP on Michelson (to act as an FTP Server), then walk over to Morley and type:

```
FTP Michelson Retrieve/c FTP.run FTP.syms
```

Alternatively, one could start an FTP server on Morley (invoking it by "FTP/O" to permit files to be overwritten on Morley's disk), then issue the following command to Michelson:

```
FTP Morley Store/c FTP.run FTP.syms
```

The latter approach is recommended for transferring large groups of files such as "*.run" (since expansion of the "*" will be performed by the Alto Executive).

To retrieve User.cm from the FTP server running on Alto serial number 123 (name unknown, but it is on the local Ethernet):

```
FTP 123'# Retrieve User.cm
```

Note that the "#" must be preceded by a single quote when included in a command line, since otherwise the Alto Executive does funny things with it. (Quotes are not necessary when typing to FTP's interactive keyboard interpreter).

To start FTP, have the FTP User connect to Maxc, and then accept further commands from the keyboard:

```
FTP Maxc
```

To retrieve <System>Pup-Network.txt from Maxc and store it on the Alto as PupDirectory.bravo, and store PupRTP.bcpl, Pup1b.bcpl, and PupBSPStreams.bcpl on <DRB> with their names unchanged:

FTP Maxc Connect/c drb mypassword Retrieve/s <System>Pup-Network.txt PupDirectory.bravo Store/c PupRTP.bcpl Pup1b.bcpl PupBSPStreams.bcpl

To retrieve the latest copy of all .RUN files from the <alto> directory, overwriting copies on the Alto disk (The single quote is necessary to prevent the Alto Executive from expanding the "*"):

FTP Maxc Ret/c <alto>'*.run

To update the Alto disk with new copies of all <alto> files whose names are contained in file UpdateFiles.cm, requesting confirmation before each retrieval:

FTP Maxc Dir/c Alto Ret/u/v @UpdateFiles.cm@

To store all files with extension .BCPL from the local Alto disk to your login directory on Maxc (the Alto Executive will expand "*.bcpl" before invoking FTP):

FTP Maxc St/c *.bcpl

To retrieve <System>Host-name/descriptor-file.txt;43 (two single quotes are necessary to get the "/" past the Alto Executive and the FTP command scanner, and one quote is necessary to get the ";" past the Alto Executive):

FTP Maxc Ret/c <System>Host-name''/descriptor-file.txt';43

To send Prog.f4, Data.f4, and Command.f4 to Fortran-Machine and then cause the FTP server on Fortran-Machine to quit (presumably to execute Prog.f4 on Data.f4 according to the commands in Command.f4):

FTP Fortran-Machine Store/c Prog.f4 Data.f4 Store/s Command.f4 Rem.cm

FTP on Fortan-Machine must be started with the /K server option switch, and Command.f4 should re-invoke FTP as its last act so that the results can be retrieved.

To release a new version of FTP, I incant:

@ReleaseAltoFTP.cm@

which the Alto Executive expands into:

FTP Maxc Connect/q Alto Store/c FTP.run FTP.syms Connect/q AltoSource Dump/c FTP.dm @ftp.cm@

and then into:

FTP Maxc Connect/q Alto Store/c FTP.run FTP.syms Connect/q AltoSource Dump/c FTP.dm @FtpSubsys.cm@ @FtpPackage.cm@ FTP.cm

and finally into:

FTP.run Maxc Connect/q Alto Store/c FTP.run FTP.syms Connect/q AltoSource Dump/c Ftp.dm Ftp.bcpl FtpNv.bcpl FtpInit.bcpl FtpInit1.bcpl FtpNvInit.bcpl FtpUserInit.bcpl FtpSubsys.decl FtpKbdInit.bcpl FtpKbd.bcpl FtpKbd1.bcpl FtpKbd2.bcpl FtpCliInit.bcpl FtpCli.bcpl FtpCli1.bcpl FtpCli2.bcpl FtpCliUtil.bcpl FtpMiscb.bcpl FtpMisca.asm FtpServerInit.bcpl FtpServer.bcpl FtpTelnetInit.bcpl FtpTelnet.bcpl FtpKeys.bcpl FtpCmdScanDsp.bcpl FtpMc.mu FtpRamTrap.mu CompileFtpmc.cm FtpSubsys.cm CompileFtpSubsys.cm CompileAltoFtp.cm LoadAltoFtp.cm MakeHiddenFtp.cm LoadHiddenFtp.cm ReleaseAltoFtp.cm CompileNovaFtp.cm LoadDosFtp.cm LoadRDosFtp.cm FtpProt.decl FtpUserProt.bcpl FtpUserProtFile.bcpl FtpUserProtMail.bcpl FtpServProtFile.bcpl FtpServProtMail.bcpl FtpPListInit.bcpl FtpPListProt.bcpl FtpPList1.bcpl FtpUtilInit.bcpl FtpUtilB.bcpl FtpUtilA.asm

FtpUtilXfer.bcpl FtpUtilDmpLd.bcpl FtpUtilCompB.bcpl FtpUtilCompA.asm BlockEq.mu
FtpOEPInit.bcpl CompileFtpPackage.cm DumpFtpPackage.cm FtpPackage.cm Ftp.cm

To load Ftp.dm from <AltoSource>, expanding it out into its constituent files:

FTP Maxc Load/c <AltoSource>Ftp.dm

To cause Memo.ears to be spooled for printing on Ears by the Maxc printing system:

FTP Maxc Store/s Memo.ears LPT:

This also works for Press files and unformatted text files if you know what you are doing. It does not do the right thing for Bravo-format files.

To use FTP as a stop-gap IFS:

FTP/T-UO

This starts only a server with overwriting of existing files permitted. When using the trident, there isn't enough space to start both a User and a Server.

6. File Property Defaulting

Without explicit information from the file system, it is often difficult to determine whether a file is Binary or Text, if Binary, what its byte-size is, and if Text, what End-Of-Line convention is used. The User and Server FTPs use some simple heuristics to determine the correct manner in which to transfer a file. The heuristics generally do the right thing in the face of incomplete information, and can be overridden by explicit commands from a human user who knows better.

The FTP protocol specifies a standard representation for a file while in transit over a network. If the file is of type Binary, each logical byte is packed right-justified in an integral number of 8-bit bytes. The byte-size is sent as a property along with the file. If the file is of type Text, each character is sent right-justified in an 8-bit byte. An EOL convention may be sent as a file property. The default is that <return> marks the end of a line.

6.1. File Types

FTP determines the type of a local file by reading it and looking for bytes with the high-order bit on. If any byte in the file has a high-order bit on, the file is assumed to be Type Binary, otherwise it is assumed to be Type Text. FTP will generate a warning, but allow you to send what it thinks to be a text file as type Binary, since no information is lost. It will refuse to send a binary file as type text.

> Don't specify a Type unless you know what you are doing. The heuristic will not lose information.

6.2. Byte-Size

If a file is type Binary, the byte-size is assumed to be 8 unless otherwise specified. The FTP User and Server will both accept binary files of any byte-size and write them as 8 bit bytes on the disk. No transformation is done on the data as it is written to the disk: it is stored in network default format. Since there is no place in the Alto file system to save the byte-size property, it is lost.

Similarly, requests for Binary files will be honored with any byte size, and whatever is on the disk will be sent to the net without transformation. Since Alto files have no byte size information, the byte-size

property will be defaulted to 8 unless otherwise specified (by the BYTE command), in which case whatever was otherwise specified will be sent as the byte size.

> Don't specify a Byte-size unless you know what you are doing. Alto-Alto transfers can't go wrong. Alto-Maxc transfers with weird byte-sizes will not work unless the byte-size specified in the Alto to Maxc direction is the same as the byte-size in which the file was stored on the Alto. If it isn't, the Alto will not give any error indication, but the result will be garbage.

6.3. End-of-Line Conventions

FTPs are expected to be able to convert text files between the local file system End-Of-Line (EOL) convention and the network convention. Conveniently enough, the Alto file system's internal representation of a text file is the same as the network standard (a bare <return> marks the end of a line). The Alto FTP does not do any transformations on text files. It will refuse to store a text file coming in from the net whose EOL convention is CRLF.

As an escape to bypass conversion and checking, EOL convention 'transparent' tells both ends NOT to convert to network standard, but rather send a file 'as is'. This is included for Lisp files which contain internal character pointers that are messed up by removing line feed characters.

> Don't specify an EOL convention unless you know what you are doing. If your text file is a Lisp source file, specify EOL convention 'Transparent'.

6.4. File Dates

The Alto file system keeps three dates with each file: Creation, Read, and Write. FTP treats the read and write dates as properties describing the local copy of a file: when the file was last read and written in the local file system. FTP treats the creation date as a property of the file contents: when the file contents were originally created, not when the local copy was created. Thus when FTP makes a file on the local disk, the creation date is set to the creation date supplied by the remote FTP, the write date is set to 'now' and the read date is set to 'never read'.

7. Abort and Error messages

Error and Abort packets are displayed in a window above the title line. Abort packets are fatal; Error packets are not necessarily so.

The most common Abort message is "Timeout. Good bye", generated when a server process has not received any commands for a long time (typically 5 minutes).

The most common Error message is "Port IQ overflow" indicating a momentary shortage of input buffers at the remote host. Receiving an Error Pup does not imply that the file in transit has been damaged. Loss of or damage to a file will be indicated by an explicit message in the User FTP window. The next iteration of Pup will probably rename 'Error Pups' to be 'Information Pups'.

8. Telnet

FTP provides a simple User Telnet as a convenience for logging into a remote host (e.g., Maxc) to poke around without having to leave the FTP subsystem and start Chat. It lacks most of the creature comforts Chat provides, such as automatic attaching to detached jobs, automatic logging in, etc. The Telnet is not enabled when the User FTP is being controlled from the command line. When the Telnet does not have

an open connection, it waits for you to type a host name with the syntax explained above for the OPEN command, and then attempts to connect to the specified host. If you wish to abort the connection attempt, hit the bottom unmarked key (opposite right-shift). You can get a larger Telnet window by not starting a server (type FTP/-S to the Executive).

9. Nova FTP

FTP is also available running under Dos Rev 4 and RDos Rev 3. Since the Nova versions are nearly identical to the Alto version (the same source files except for initialization), only the differences are listed here.

1) Ignore all references to display windows. All printout goes to device #11, whatever that is.

2) Ignore all references to 'unmarked keys' such as for aborting connection attempts and directing keyboard input to various windows.

3) Lack of memory and lack of a windowing display made including a Telnet impractical on the Nova.

4) The syntax of the command line is limited to that acceptable to the Nova operating system. Warning: the command line examples given above may not all work on a Nova.

5) The Nova OS does not maintain a username or password, so all interactions with a Maxc system or an IFS will require the user to supply them.

6) File creation dates are not supported, so there is no Update option to RETRIEVE, and the LIST command does not show dates.

9.1. FTP releases

The Nova FTP subsystem consists of a save-file, FTP.SV, and an overlay-file, FTP.BB. You must get BOTH files when a new version of FTP is released. If you rename FTP.SV you must rename FTP.BB to have the same name (for instance if you rename FTP.SV to be OLDFTP.SV you must also rename FTP.BB to be OLDFTP.BB). New releases of FTP will be distributed as dump files with a consistant pair of save- and overlay-files.

9.2. Device codes

FTP assumes that Nova Ethernet interfaces have device codes 73 and 74, 63 and 64, or 53 and 54. It will use all interfaces with these codes that seem (from reading some status registers) to be Ethernets. The Dos version of FTP assumes that Nova MCA interfaces are device code 6 and 7, or 46 and 47. It will use all interfaces with these codes that seem to be MCAs.

9.3. RDos notes

FTP is big, and will not run under some RDos systems. If you have trouble, generate a smaller system and boot from it when running FTP. FTP disables parts of RDos with patches which may not work for versions other than Rev 3. It will NOT work under an RDos that uses the memory map hardware. The RDos version does not include MCA drivers.

10. Revision History

April 1976

First release.

May 1976

/Q switch added to CONNECT. Connection requests to the User FTP and Telnet can be aborted. Login prompt changed. 1 minute Timeout added when waiting to finish after a command line error. User FTP automatically recovers from more "No" responses from the remote server.

June 1976

Dos version released. DIRECTORY and LIST, commands added. Update (/U) option added. File creation dates added. 5 minute no-activity timeout added to FTP Server. FTP version, time-of-day, and machine address added in top window. "Ding" now flashes only the affected window instead of the whole display.

August 1976

RDos version released. Same as June release for Dos and Alto.

October 1976

DUMP and LOAD commands added to user FTP. KILL command added. Free disk page count added to the title line. Verify (/V) switch added to the RETRIEVE command.

November 1976

Bug fixes to the October release.

May 1977

This version was only released to friends. KILL command removed and turned into a server option. DEBUG command moved into new USER and SERVER commands. Trident disk option (/T) added. User LIST command improved and Server LIST response implemented. Password checking by the FTP server implemented. Telnet window enlarged at the expense of possibly losing information from the top of the window if the lines are very full. DELETE, RENAME, and DEVICE commands implemented. Much internal reorganization so that the protocol modules could be used in IFS and released as a package.

July 1977

Global switches changed. <Shift-Swat> should work more reliably now. User LIST command further improved. Keyboard command interpreter is much more robust and consistant. Command line STORE and DUMP go much faster since they look up files using MDI. FTP/Tx opens Trident unit 'x'. LOGIN command added to command line interpreter.

November 1977

Special microcode added to speed up execution.

March 1978

User log option added (see /L and /A switches and 'FTP User Log' section). AllocatorDebug switch removed. New command line commands COMPARE, OPEN, and CLOSE added. Command line errors are handled differently (see /E global switch and 'Command Line Errors' section). When using a Trident, either a User or a Server FTP is started but not both (see the section on Trident disks).

September 1979

This is a maintenance release coordinated with OS17, fixing a few bugs and reloading with current packages. CONNECT cancels any previous DIRECTORY. CLOSE cancels any previous CONNECT, DIRECTORY, DEVICE, TYPE, BYTE, or EOLC.. Multiple logical file systems on a T-300 can now be addressed: Ftp/T400 opens logical filesystem 1 on physical unit 0.

ListSyms - a subsystem for listing Syms files

The ListSyms subsystem takes a Syms file (produced by BLDR) and converts it to a useful human-readable form. ListSyms produces a file with several parts:

A listing of the space occupied by each binary output file (.Run or .BB).

A listing similar to the listing optionally produced by BLDR, i.e. a list, sorted by BR file and location within the file, of all static symbols defined, with an indication as to whether the symbol is external and whether it is a procedure, label, or static variable.

A list of all statics in alphabetic order, accompanied by the name of the BR file in which each one is defined and (optionally) a list of all the BR files in which each is used.

A list similar to the preceding, but listing the statics for each file separately, and only listing statics declared external (i.e. accessible from other files).

A concordance of undefined externals: for each BR file which references undefined externals, it lists those externals in alphabetic order under the file name.

One invokes ListSyms as follows:

```
>ListSyms inputfile outputfile
```

Inputfile will normally be something.Syms: if it has no extension, ListSyms will supply .Syms. Outputfile may be omitted, in which case ListSyms will take inputfile (shorn of extension if any) and append .BZ to form the output file name.

ListSyms accepts 7 switches, all global:

```
/A produces the alphabetic listing
/F produces a file-by-file alphabetic listing with cross-reference
/N produces the numeric (file-by-file) listing
/O produces only the listing of the binary file sizes
/S includes static variables, which are normally omitted
/U produces the listing of undefined externals
/X produces the alphabetic listing with cross-reference
```

The switches may be either upper or lower case, and /S is independent of the other switches. If none of /A, /F, /N, /O, /U, or /X appears, you will get the /A, /N, and /U listings but no cross-reference.

ListSyms starts by printing a message of the form

```
ListSyms of [date] -- [inputfile] -> [outputfile]
```

If ListSyms completes normally, it will print a message of the form

```
12345b characters written on outputfile
```

ListSyms produces a variety of error messages. Currently these are:

```
[filename] does not exist
```

indicates ListSyms was unable to open the Syms file.

```
Syms file too big
```

indicates insufficient room for reading the Syms file. ListSyms aborts.

```
Can't open [filename]
```

ListSyms was unable to open the outputfile or one of the BR files required for /U or /X. In the former case, ListSyms aborts; in the latter, it continues.

```
[filename] is not a proper BR file
```

One of the BR files mentioned in the Syms file does not have the proper format. ListSyms ignores the file and continues.

```
[filename] is too big to process
```

One of the BR files was too big to read in. ListSyms ignores it and continues.

```
Too many BR files
```

There were too many BR files to process in the available memory. ListSyms aborts.

```
No room for bit table
```

There was not enough room to hold the bit table used for /U or /X (or /A if any undefined symbols were present). ListSyms aborts.

ListSyms is quite fast: it processes BRAVO.Syms in about 20 seconds, and a typical modest program takes less than 10 seconds.

MailCheck

This simple subsystem attempts to check for mail for a user at some other host (e.g. Maxc) via the Ethernet. It displays one of the following messages:

? This Alto has no Ethernet interface!
? Can't find a host named '<host>': <error message>
? No response from <host>
? <user> not valid user at <host>: <error message>
? Error: <pup error message>
New mail for <user> on <host>: <date> <sender>
No new mail for <user> on <host>

Various options can be controlled by switches and/or by an entry in your User.Cm.

Valid switches are:

/1	Check mail on Maxc1 (default).
/2	Check mail on Maxc2.
<host>/H	Check mail on <host>.
<user>/U	Check mail for <user> (default is the user name obtained from the Alto operating system).
/R	If there is new mail, execute a command line when MailCheck exits. The command line defaults to "@READMAIL.CM@", i.e. to execute the contents of the file READMAIL.CM as a command, but this can be changed in the User.Cm as outlined below.

In addition, if there may be a section in your User.Cm labeled [MAILCHECK] with the following possible entries:

HOST: <host>	Sets the default host to check.
USER: <user>	Sets the default username to check.
NEWMAIL: <string>	Sets the command line to be executed if there is new mail. Within the command line, the host name is substituted for "@H" and the user name for "@U"; to put an "@" in the command line it is neccessary to put two in the string.

For example, you might add the section:

```
[MAILCHECK]
HOST: Maxc2
NEWMAIL: CHAT @H MSG.DO/D
```

Where MSG.DO is a file on your alto disk which contains "MSG<return>".

One useful option is to put Mailcheck.Run inside the eventBooted section of your USER.CM, so that Mailcheck will be run whenever you boot, e.g.

```
[EXECUTIVE]
eventBooted: Mailcheck.Run // eventBooted
eventRFC: FTP/OK //eventRFC
...
eventClockWrong: SetTime // eventClockWrong
```

Updates: As of March 1978, Mailcheck no longer does a SetTime

MoveToKeys

The Alto can boot-load a file beginning at any legal disk address. The disk address is supplied by holding down a collection of keys simultaneously while pressing the boot button. The MoveToKeys subsystem simplifies the task of getting a .boot file to begin at a specified physical disk location. To invoke MoveToKeys, type:

MoveToKeys filename keylist

to the Alto Executive. "filename" is the name of the file whose first page (technically, page 1, not page 0) is to be moved to the disk address corresponding to "keylist". The legal keys are 5, 4, 6, 7, D, E, K, P, U, V, 0, /, and . (Remember, to type a "/" to the Alto Executive, you must quote it.) A typical use of MoveToKeys is:

MoveToKeys Dumper DU

The file Dumper.boot could then be boot-loaded by holding down the D and U keys while pressing the boot button.

MoveToKeys will prompt for parameters omitted from the command line and will complain if any of the parameters supplied are illegal. (For example, not all subsets of the set of legal keys correspond to legal disk addresses.) In addition, the global switch /V ("verbose mode") will give you detailed information about the pages MoveToKeys manipulates.

MoveToKeys actually works by determining what page resides at the specified disk address and swapping it with page 1 of the specified file. Depending upon the pages involved, MoveToKeys must patch up various pointers within the Alto file system to ensure a consistent representation of files and directories. (A previous version of MoveToKeys did not do this correctly in all cases.)

Mu: Alto Microassembler

This document describes the source language and operation of Mu, the Alto microcode assembler. Mu is downward compatible with Debal, the original Alto assembler/debugger, but has a number of additional features. Mu is implemented in BCPL, and runs on the Alto.

1. The source language

An Alto microprogram consists of a number of statements and comments. Statements are terminated by semicolons, and everything between the semicolon and the next Return is treated as a comment. Statements can thus span several text lines (the current limit is 500 characters). All other control characters and blanks are ignored. Bravo formatting is also ignored.

Statements are of four basic types: include statements, declarations, address predefinitions, and executable code. The syntax and semantics of these constructs is as follows:

1.1. Include Statements

Include statements have the form:

filename;

They cause the contents of the specified file to replace the include statement. Nesting to three levels is allowed.

1.2. Declarations

Declarations are of three types: symbol definitions, constant definitions, and R memory names.

1.2.1. Symbol Definitions

Symbol definitions have the form:

$name$I n_1,n_2,n_3;

The symbol "name" is defined, with values n_1, n_2 and n_3. There is a standard package of symbols for the Alto (AltoConstsxx.Mu, where xx is the current microcode version) which should be 'included' at the beginning of every source program. For those who must add symbol definitions, the interpretation of the n's is given in the appendix.

1.2.2. Constant declarations

Normal constants are declared thus:

$name$n;

This declares a 16 bit unsigned constant with value n. The assembler assigns the constant to the first free location in the constant memory, unless the value has appeared before under another name in which case the value of the name is the address of the previously declared constant.

An alternative constant definition is used for mask constants which have a specified bus source field (recall that the constant memory address is the concatination of the rselect and bus source fields of the microinstruction). The syntax is:

$name$Mn:v; $4 \le n \le 7$, $0 \le v < 2**16$

N specifies the desired bus source value, v is the constant value.

1.2.3. R Memory declarations

R memory names are defined with:

$name$Rn; $0 < n < 40B$
(100B if your Alto has a RAM board, as most do)

An R location may have several names.

1.3. Address predefinitions

Address predefinitions allow groups of instructions to be placed in specified locations in the control memory, as is required by the OR branching scheme used in the Alto. Their syntax is:

!n, k, $name_0$, $name_1$, $name_2$, ... , $name_{k-1}$;

This declaration causes a block of k consecutive locations to be allocated in the instruction memory, and the names assigned to them. n defines the location of the block, in that if L is the address of the last location of the block, L and n = n. Usually, n will be 2**p-1 for some small p. For example, if the predefinition

!3, 4, foo0, foo1, foo2, foo3;

is encountered in the source text before any executable statements, the labels foo0-foo3 will be assigned to control memory locations 0-3. If there are too few names, they are assigned to the low addresses in the block. If there are too many, they are discarded, and an error is indicated. If there are missing labels, e.g. "foo0,,foo2,;", the locations remain available for the normal instruction allocation process. A predefinition must be the first mention of the name in the source text (forward references or labels encountered before a predefinition of a given name cause an error when the predefinition is encountered.)

A more general variant of the predefinition facility is available. The syntax is:

%mask2, mask1, init, L_1, L_2, ... L_n;

The effect of this is to find a block of instructions starting at location P, where P and mask1 = init, and assign the L's to 'successive' locations under mask2. For example:

%1, 1, 0, x0, x1;

would force x0 to an even instruction, x1 to odd (the normal predefinition for most branches).

%360, 377, 17, L0, L1, ... L15;

Would place L0 at xx17, L1 at xx37, L2 at xx57, etc.

As before, if there are unused slots (e.g., 'L12,,L14') they are available for reassignment, and MU complains if there are too many labels for the block.

1.4. Executable statements

Executable code statements consist of an optional label followed by a number of clauses separated by commas, and terminated with a semi-colon

label: clause, clause, clause, ...;

If a label has been predefined, the instruction is placed at the control memory locaion reserved for it. Otherwise, it is assigned to the lowest unused location.

Clauses are of three types: gotos, nondata functions, and assignments.

Goto

Goto clauses are of the form ':label', and cause the value of the label to be assembled into the Next field of the instruction. If the label is undefined, a chain of forward references is constructed which will be fixed up when the symbol is encountered as a label.

Nondata Functions

Nondata functions must be defined (by a literal symbol definition) before being encountered in a code clause. This type of clause assembles into the F1, 2, or 3 fields, and represents either a branch condition or a control function (e.g. BUS=0, TASK).

Data transfers (assignments)

All data transfers are specified by assignments of the form:

$$dest_1 \leftarrow dest_2 \leftarrow \ldots \leftarrow source$$

This type of clause is assembled by looking up the destinations, checking their legality, and making the field assignments implied by the symbol types. Each destination imposes definitional requirements on the source (e.g., ALU output must be defined, Bus must be defined). These requirements must be satisfied by the source in order for the statement to be legal.

When the source is encountered, it is looked up in the symbol table. If it is legal and satisfies the definitional requirements imposed by the destinations, the necessary field assignments are made, and processing continues. If the entire source defines the Bus, and the only remaining requirement is that the ALU output must be defined (e.g., L←MD), the ALUF field is set to 0 (ALU output = Bus), and processing continues.

If neither of the above conditions holds, the source can legally be only a bus source concatenated with an ALU function. The source token is repeatedly broken into two substrings, and each is looked up in the symbol table. If two substrings can be found which satisfy the requirements, the field assignments implied by both are made; otherwise, an error is generated. This method of evaluation is simple, but it has pitfalls. For instance, L←2+T is legal (providing that the constant "2" has been defined) but L←T+2 is not (the Bus operand must always be on the left). Note that 'L←foo+T+1' specifies a bus source of 'foo' and an ALU function of '+T+1'.

CAVEAT: The T register may be loaded from either the Bus or the output of the ALU, depending on the ALU function. The assembler does not check to see whether an assignment of the form 'T←ALU' specifies an ALU function that actually loads T from the ALU. For example, the clause 'L←T←MD-1' is accepted, but its effect is to load T directly from MD. If this is what you intend, it makes matters clearer if you write 'L←MD-1, T←MD'; if it is not what you intend, you are in trouble. Beware!

The constant "0" is special, in that when one or more clauses in a statement require that the bus be 0, generation of the constant is deferred until the end of the statement. At that point, if any clause

has caused the R memory to be loaded, the constant is not used, since the hardware forces the bus to 0 in this case.

The destination "SINK" allows a clause to specify a bus source without specification of a destination. It is useful, for example, in constructs of the form 'SINK←AC0, BUS=0', which puts AC0 on the bus to be tested by the nondata function 'BUS=0'. You can also write things like 'SINK←mask constant, L←DISP XOR T', which will cause the value of DISP to be anded on the bus with the mask constant.

2. Operation

The assembler is invoked with:

MU/global-switches sourcefile listfile/L binfile/B statfile/S

Legal global switches are:

/L produce a listing file
/D debug mode
/N do not produce a binary file (overridden by binfile/B)

If listfile/L is absent but the /L global switch is set, listing output will be sent to sourcefile.LS.

If binfile/B is absent, binary output is sent to sourcefile.MB.

If statfile/S is absent, statistics for the assembled program are appended to the listing file if there is one; otherwise, no statistics are generated. The default extension for a /S file is '.Stats'.

The default extension for sourcefile is '.Mu'.

Error messages will be sent to the listing file if one has been specified, unless debug mode has been set. In debug mode, errors are sent to the system display area, and a pause occurs at at every error (and at certain other times). Typing any character proceeds.

If no listing file has been requested, debug mode is set independent of the global switch.

3. Output file

The assembler produceds Micro format binary output. The string names of the two memories specified in the file are CONSTANT and INSTRUCTION. Only defined locations in these memories are output. Micro format is compatible with the PRom blowing program, the RamLoad program, and the PackMu/LoadRam software. Note that the instruction memory specified in the binary file does not include the 3 bit F3 field, which exists only in the debugging RAM.

4. Listing file

The listing file contains:

1.) All error messages (unless debug mode is set)

2.) A listing of all unused but predefined locations and unresolved forward references.

3.) Two listings of the contents of the constant memory, the first sorted by address and the second by value.

4.) A listing of the names assigned to the R memory

5.) A listing of the object and source code (with comments and declarations removed. The 35 bit instruction is printed out in the following order:

Location: RSel, ALUF, BS, F1, F2, LoadL, LoadT, F3

6.) The microprogram statistics (unless sent to a separate file).

Appendix I: Literal symbol definitions

The value of a symbol is a 3 word quantity. The first word contains a type (6 bits) and a value (10 bits) which detemines the interpretation of the symbol in all cases except when it is encountered as the source in a data transfer clause (assignment). The second word contains the type and value used in this case.

The third word contains bits specifying the definitional requirements and source attributes applied when the symbol is encountered in an assignment. The definitional requirements are represented by single bits, where zero means 'must be defined' and one means 'don't care'.

Bit 0: 0 if L output must be defined	(destination-imposed requirements)
Bit 1: 0 if BUS must be defined	"
Bit 2: 0 if ALU output must be defined	"
Bits 3-7: Unused (?)	
Bit 8: L is defined	(Source attributes)
Bit 9: Bus is defined	"
Bit 10: ALU output is defined	"
Bit 14: ALU output is defined if BUS is defined	

Assignment processing proceeds by ANDing together the attribute words for all the destinations. The result contains zeroes in bits 0-2 for things that must be defined and ones elsewhere.

When the source token is encountered, if it is a defined symbol it is tested by checking the definitional requirements of the destinations against the corresponding attributes in the source. If all destination requirements are satisfied, the clause is complete. If the only unsatisfied requirement is ALU definition, and if the Bus is defined, the ALU function is set to gate the bus through (thereby defining the ALU), and the clause is complete. If this doesn't work, or the source token is not a defined symbol, the source string is dismembered in a search for two substrings, the first of which defines the Bus (bit 9), and the second of which defines the ALU output if the Bus is defined (bit 14). If two substrings are found, the implied assignments are made, and the clause is complete. Otherwise, an error is indicated.

The symbol type(s) determine the fields to be set in the microinstruction: Some types are legal only as an isolated clause, some are legal only as the source or destination in an assignment. The currently defined types are:

Type:	Legal as:	Instruction Field Receiving Value:	Side Effects:
0 Illegal	never		
1 Undefined address	address		
2 Defined address	address	Next	
3 R location←	destination	RSel	Defines Bus to be 0
4 ←R location	source	RSel	
5 ←Constant	source	RSel, BS	
6 Bus source	source	BS	
7 Non-data F1	clause	F1	
10 F1←	destination	F1	
11 ←L defining F1	source	F1	(←L LSH 1, etc.)
12 Non-data F2	clause	F2	
13 F2←	destination	F2	
14 ←Data F2	source	F2	BS←1, RSEL←0 (←DNS, ←ACDEST)
15 Data F2←	destination	F2	BS←0, RSEL←0 (ACDEST←, ACSOURCE←)
16 END	clause	-	Not used by Mu.
17 ←L	source	-	
20 L←	destination	LoadL	

21	Non-data F3	clause	F3	
22	F3←	destination	F3	
23	←F3	source	F3	
24	←ALU functions	source	ALUF	
25	T←	destination	LoadT	
26	←T	source	ALUF	ALUF←1
27	No longer used			
30	Predefined address			
31	←LMRSH, ←LMLSH	source		
32	←Mask constant	source		
33	←F2	source	F2	BS←2
34	←F1	source	F1	BS←2
35	XMAR←	destination	F1, F2	F1←1, F2←6

The current symbol definitions are contained in file AltoConsts23.Mu.

5. Revision History

October 24, 1974

'%' predefinition facility added.

March 4, 1975

This version has changed from previous releases in that the .BM file contains micro format type 5 blocks which contain address symbols for the constant, instruction, and R memories. Programs which read these files will be expected to deal with this type of block.

October 11, 1977

Bugs fixed: garbage in listing if statement too long; occasionally scrambled R-register listings; premature termination at the end of 'insert' files.

Features: longer statement buffer (500 characters); symbol type 35 for XMAR←; '.Stats' file generated conditionally; checks for loading S-register from shifter; reports length in octal and decimal; strips Bravo formatting.

March 25, 1978

Bug fixed: leaving the semicolon off the end of a predefinition yielded erroneous results with no error message.

Features: listing file contains constants sorted by value as well as by address; source filename extension defaults to '.Mu'.

Network Executive

NetExec is an Alto command processor for invoking certain subsystems via the Ethernet without using the local disk. It is useful for rebuilding a smashed disk and for loading diagnostic programs when the disk is sick. Its user interface is intentionally similar to the standard Alto Executive.

The program is invoked by holding down the <backspace> and <quote> keys while pressing the boot button. You must continue to hold the keys down until a small square appears in the middle of the screen, then you can let go. NetExec and all of the programs invoked by it are boot-format files kept by 'boot-servers' -- programs which implement the Alto boot protocol. Most gateways and some other programs (such as Peek) contain boot-servers.

When the NetExec arrives, it displays a ">" and blinks its cursor to indicate that it is ready for commands from the user. In parallel with this it displays a pair of lines near the top of the screen with its name and version number, a digital clock, and the machine's internetwork address.

Typing "?" causes the NetExec to display a list of the boot-files it knows how to invoke. NetExec builds this list by probing the network for boot servers and asking them what boot files they are willing to give out. There are also some built-in functions which are listed by "?" as if they were boot files:

Probe	Causes NetExec to probe the network looking for boot servers. If it discovers any new ones, it will add the new boot files to its list. This is done once automatically when NetExec starts.
SetTime	Causes NetExec to probe the network looking for a time server. If it discovers one, it sets the Alto's clock from it. This is done once automatically when NetExec starts.
FileStat	Prompts you for a boot file name and tells you all about it: its boot file number, the host from which the NetExec will obtain it, and the key combination which will boot it directly.
Quit	Boots DMT

In the future, common subsytems should be stored in a few places throughout the network, not on every local disk; perhaps the local disk can be eliminated entirely. Doing so requires a much better integration of network and OS facilites than currently exists. The NetExec described here is not intended to do this. There are several limitations in the current implementation:

1) Most boot-files are core images and so are quite large. Typical boot-servers have space for about 15 core-image files.

2) Boot-files are not properly hooked into the local disk. Programs which use overlays or keep internal file pointers (such as Bravo and DDS) will not work.

3) Boot-servers typically run in machines with some other primary purpose, such as gateways, and must not consume too many resources. As a result, booting is slow and only one machine can be served at a time.

OEDIT

The OEDIT program is for looking at and modifying Alto files and Alto Trident files, in octal and other output formats. Call it with OEDIT f1 f2 ... where the f's are the names of the files you want to look at. It will display the contents of the corresponding words of all the files on the same line. There is a limit of four files which can be looked at simultaneously. If you want to be able to modify the first file, use the/W switch on the OEDIT command. If you don't use this switch, OEDIT will request confirmation before letting you write into any of the files.

When it starts, the program computes the length (in bytes) of all the files. For large files this can take upwards of 15 seconds, so don't be alarmed by the delay.

After typing the lengths, OEDIT waits for commands:

n/	show location n of each file
lf	show the next location of each file
↑	show the previous location of each file
cr	show the current location again
n!	show locations n to n+37 of each file
>	show the next 40 locations of each file
<	show the previous 40 locations of ech file
nF	beginning at current location in the first file, find a word containing n, show it and its address
Q	quit

The lf, ↑, <, >, and cr commands can be preceded by a number which is written into the current location of the first file.

All numbers are octal, except in certain output formats described below. All addresses are word addresses (even though the lengths are shown in bytes.) Unless directed otherwise, Oedit shows each value as an octal number, two octal bytes, and two Ascii characters. One can suppress these output formats, and can select additional formats, using global switches. The following table specifies the output formats and corresponding global switches:

O	displays a full-word octal value
H	two octal bytes
A	two ASCII bytes
X	two hexadecimal bytes
E	two EBCDIC bytes
D	a full-word signed decimal value
N	two decimal bytes
-	the output format specified by the next letter will not be used

Thus, the standard default display is obtained using "OEDIT/OHA filename". If one wanted to display the file in hexadecimal, ASCII, and EBCDIC only, one would type "OEDIT/-O-HXE filename".

To examine and modify Trident files, use the global T switch: e.g., "OEDIT/T ... ".

Alto microcode overlays

Large systems which use the Alto control RAM, such as ByteLisp and Mesa, inevitably want to put more instructions in the RAM than will fit. When this happens, the system implementors can choose either to implement the additional functions in software, or to change the contents of the RAM dynamically. The package described here provides for relatively cheap dynamic overlaying of the RAM. The overlay regime can be very simple (just one overlay in RAM at a time) or complex (a nested allocation scheme) with no changes in the swapper or the overlays themselves.

Users of this package must, of course, still decide when loading microcode is preferable to falling back into Nova code. In terms of space, one microinstruction does about 2/3 as much work as a Nova instruction, and takes 32 bits rather than 16, so (overlaid) microcode takes about 3 times as much core space for equivalent tasks. The package presented here imposes an additional space overhead which may amount to as much as 2 * the square of the number of overlays. In terms of speed, loading a microinstruction takes about as long as executing a Nova instruction, and the package described here adds an additional time roughly equal to 1 Nova instruction for each overlay each time a new overlay must be loaded, so for totally straight-line code the net execution time favors Nova implementation by about a factor of 2 (i.e. to break even, a given overlay must be executed at least twice). However, microcode has easy access to the state information stored in the processor's R registers, while Nova code does not (unless it can all be passed through the AC's), so this may make microcode execution preferable even in the case of straight-line code executed only once.

1. How to use it

Using microcode overlays requires three steps that differ from normal use of the RAM. The Mu assembly process is different; the Oram program must be run to construct the data structures necessary for the swapper; and a small amount of extra initialization is required at runtime.

The first step in constructing overlayable microcode is to decide how to break up one's microcode into overlays and to identify the entry points to each overlay. (One overlay may have more than one entry point.) The microcode sources must be broken up into files: a main file that includes all the resident code, plus predefinitions (but no code) for all entry points of all overlays; an initialization file (to be described in a moment) that supplies dummy code for all entry points; and files for the individual overlays.

The main file must include the following code at the beginning:

```
!0,1,zero;     Required by the swapper
$ramvec2$Rnn;  An S register for the base of the overlay table

[other predefinitions, symbol defs, constants, registers, etc.]

#swapper.mu;   The swapper
```

This code must occur at the beginning of the main file because the swapper's entry point (label "swapper") must be predefined as location 1000 in the RAM.

The initialization file must have the following form:

```
#main.mu;      (or whatever the main file is called)

ent0:  T ← 0, :swapper;
ent1:  T ← 1, :swapper;
ent2:  T ← 2, :swapper;
ent3:  T ← 3, :swapper;
```

and so on for all the entry points. (Ent0, etc. should be replaced by the names of the entry points, of course.)

Since microcode is not relocatable in the RAM, all decisions about what overlays can be co-resident must be made at assembly time.

After assembling the dummy file and each leaf overlay file with Mu in the usual way, run the Oram subsystem as follows:

```
>Oram xx.BR init.MB ov1.MB ... ovm.MB
```

where xx.BR is the BR file on which Oram will write the overlay tables, init.MB is the result of assembling the initialization file, and ov1.MB through ovm.MB are the results of assembling the leaf overlay files. If all goes well, Oram will produce a variety of messages ending with

```
nnn words written on xx.BR
```

and return to the Executive. Oram also writes all its messages on a file called Oram.Lst.

When you load your program with Bldr, you must include the file xx.BR produced by Oram. The data in this file, unlike the initial RAM image produced by PackMu, is required throughout the running of your program. You must also load the RWREG library package to obtain the WriteReg procedure used below, but this is only needed during initialization.

When loading the RAM during initialization, your program must include the following code:

```
...
external [ MCbase; MCtop ]     // defined in xx.BR
if (MCbase&1) ne 0 then
[ let len = @MCtop
  MoveBlock(MCtop-len-1, MCtop-len, len)
  MCbase = MCbase-1
]
WriteReg(nn, MCbase-2)
...
```

where nn is the register number in the definition of ramvec2 in the main file.

2. Design details

In the RAM, the entry instructions of each overlay are all in the permanently resident code. If the overlay is present, the entry instruction is just the first instruction of its code; in this case we say the entry instruction is "valid". If the overlay is absent, the entry instruction loads T with the entry number and branches to the swapper (the entry instruction is "invalid"). Thus when an overlay is loaded, the entry instructions of all overlays it overlaps must be invalidated. The chief advantage of this approach is that there is absolutely no time overhead if the overlay is already in the RAM, so it is feasible to overlay very short sequences (15 instructions, say).

There is just one global data structure (in core) that describes the overlay structure: a table indexed by 2 * entry number which points to overlay descriptions, described in the next paragraph, and also specifies where to start execution after the overlay is loaded. (This arrangement permits a single overlay to have multiple entry points.) The origin of this table is the only thing known to the swapper.

The description of an overlay (in core) must begin at an even location, and has two parts:

1) An invalidation table which specifies how to overwrite entry instructions. Each entry in this table is a 2-word object: the first word is a RAM address, the second word is the upper half of the microinstruction to write there (the lower half always being "BUS←constant, Load T, branch to swapper"). The last entry is flagged by having bit 0 of the RAM address set.

2) A sequence of instruction blocks. Each block begins with a 2-word header (100000b+RAM address, 0). The following data are a sequence of instructions where each instruction's NEXT field specifies where to load the following one: this sequencing scheme eventually requires the block to end. This sequence is terminated by a final block consisting of two zero words.

The swapper is a routine in the resident microcode which expects an entry number in T, loads the appropriate overlay, and branches to the entry. It must fetch the overlay's description from core and then do the following things:

1) Invalidate the entry instructions of all overlays with which the one being loaded conflicts.
2) Load the code, which must include the entry instructions specified as being newly valid;
3) Branch to the code. The initial RAM load must have all entry instructions invalid.

3. Mu/Bldr interface

The third design issue is how best to get the necessary data structures incorporated into Bcpl/Nova programs. It turns out that it is possible to support nested overlays with no changes to Mu. For example, suppose that the main body of the microcode is M. and that we have three overlays: X (entry point X1), which takes all the overlay space, and Y (entry points Y1 and Y2) and Z (entry point Z1), which will both fit at the same time. Assemble the following configurations with Mu: M+X, M+Y, and M+Y+Z. Then an overlay preparation program, Oram, can compute all the necessary tables and produce a .BR file that can be loaded with the user's program.

It is necessary to be a little careful to arrange that the entry instructions fall in the same locations in all assemblies. Furthermore, if it is desired that one routine occupy a subset of the RAM locations of another, they must have the same configuration of predefinitions (and, of course, appear at the same place in the assembly sequence). Here is a sketch for the example:

```
M contains (somewhere):
    !0,1,X1;
    !0,1,Y1;
    !0,1,Y2;
    !0,1,Z1;

X contains:
    X1:   [code for X]

Y contains:
    Y1:   [code for Y]
    Y2:   [more code for Y]

Z contains:
    Z1:   [code for Z]
```

In general, some of the predefinitions could be omitted if the entry addresses were to be predefined earlier, for example if they were entries in some kind of opcode dispatch. In addition, there must be another file W which is assembled with M to produce the initial RAM load:

```
W contains:
    X1:   T ← 0, :swapper;
    Y1:   T ← 1, :swapper;
    Y2:   T ← 2, :swapper;
    Z1:   T ← 3, :swapper;
```

The pointer table would have the appearance

```
        Xdesc; X1;
        Ydesc; Y1;
        Ydesc; Y2;
        Zdesc; Z1;
```

and the individual descriptions would be as follows:

```
    Xdesc:  Y1;     invalidate Y and Z
        BUS←1 (hi part);
        Y2;
        BUS←2 (hi part);
        #100000+Z1;
        BUS←3 (hi part);
        [code for X]
        0;
        0;
    Ydesc:  #100000+X1;     invalidate X
        BUS←0 (hi part);
        [code for Y]
        0;
```

```
        0;
Zdesc:  #100000+X1;     invalidate X
        BUS←0 (hi part);
        [code for Z]
        0;
        0;
```

Fortunately, given the .MB files, the Oram subsystem can construct all the tables itself. Oram assumes that any instruction in the base file (W) which branches to the swapper is an entry instruction.

PackMu, Rpram, ReadPram

These two subsystems and one library package make it easy for Alto programs which use the RAM to check the constant memory and load the RAM as part of their initialization. The first subsystem, PackMu, takes the output of Mu (a .MB file) and converts it to a "packed RAM image" which is easy to load. The second subsystem, Rpram, reads a packed RAM image, checks the constant memory, and loads the RAM (i.e., it is a microcode loader). This function is also available through a pair of library routines ReadPackedRAM and LoadPackedRAM (available on a file called ReadPram.bcpl).

A packed RAM image is a .BR file containing 4401b words of data. The first word is ignored. The next 400b words are the desired contents of the constant memory: a zero word (which Mu cannot generate) means "don't care". Constant 0 is reserved for a version number, to help programs check that they are getting the correct RAM contents. The remaining 4000b words are the contents of the RAM. Each instruction occupies two words, first high-order part, then low-order part, e.g. words 0 and 1 go into RAM location 0, words 2 and 3 into RAM location 1, and so on.

The invocation format for PackMu is

```
>PackMu foo.MB foo.BR version staticname
```

Foo.MB is the output from MU. Foo.BR is the file for the packed RAM image. Version (optional) is a RAM version number which will be written as constant 0 in the output file; if omitted, it defaults to zero. Staticname (optional) is the name for the static in foo.BR which will point to the RAM data; if omitted, it defaults to RamImage. PackMu prints out

```
xxx constants, yyy instructions
```

to indicate the number of constants and instructions read from foo.MB. If foo.MB is somehow illegal, PackMu prints

```
Error:
```

and an error message instead.

The invocation format for Rpram is

```
>Rpram foo.BR version rambank
```

where foo.BR is the output from PackMu and rambank is the bank number (1, 2, or 3) if Alto has the 3K RAM option. If there are any disagreements between the constants in foo.BR and the actual constant memory, Rpram prints

```
Constant nnn is xxx, should be yyy
```

for each constant that disagrees, and a summary message

```
nnn constants differ
```

at the end of loading (but it still loads the RAM). If version is supplied and disagrees with constant location 0 in foo.BR, Rpram prints

```
RamVersion in file is xxx; version expected is mmm
```

If Rpram believes that foo.BR is not a file written by PackMu, it prints

```
Bad RAM image
```

If everything is OK, Rpram prints nothing.

To read in a packed RAM image file from a program, use the subroutine ReadPackedRAM(stream, lvRamV [], rambank [1]). The stream argument should be a word-item input stream positioned at the beginning of a foo.BR file; lvRamV, if supplied, is taken as the address of a variable in which to store the value given by the file for constant 0 (i.e. the RAM version). ReadPackedRAM does exactly the same thing as the Rpram subsystem, including printing disagreement messages on the display, but instead of printing the summary message it just returns the number of disagreements, or -1 in the case of a bad RAM image file. Rpram essentially just opens foo.BR and calls ReadPackedRAM.

Alternatively, you may wish to load the RAM image foo.BR with your program. In this case, use the subroutine LoadPackedRAM(staticname, lvRamV [], rambank [1]) where staticname is the name you gave to PackMu. LoadPackedRAM does the same thing as ReadPackedRAM, except it takes the data out of memory instead of from a file.

On Altos with the 3K RAM, note that since LoadPackedRAM and ReadPackedRAM use two words

in RAM bank 1 for checking the constant memory, you should load bank 1 last if you have a multi-bank microprogram.

Maintainer's notes:

PackMu uses the library packages GP and ReadMu.

Rpram uses the library package GP.

PeekPup

PeekPup is a small subsystem enabling one to peek at Pups going to and from a particular Ethernet host. It is intended as an aid in debugging new Pup software.

PeekPup is invoked by the command

PeekPup hostnumber filename

where "hostnumber" is the Ethernet address (octal) of the host whose packets you want to spy on and "filename" is the name of a file to write the output on. The program then looks for packets whose Ethernet source or destination address is equal to "hostnumber", and buffers them in memory. For each Pup so processed, "!" is displayed on the screen. PeekPup terminates when any key is pressed, at which point it interprets the last 200 Pups received and writes the result on the specified file.

The output is mostly self-explanatory. The numbers in the left margin represent a millisecond clock (with no particular starting value and wrapping around at 32768). For each Pup, a few lines of output are generated; the information about Pups sent to the host being spied upon is indented further than information about Pups generated by that host. Pup headers are fully interpreted, and Pup contents are displayed as either text or a series of octal numbers representing bytes; large Pups get only the initial portion of their contents displayed, followed by "...".

Pressedit

Pressedit is useful for combining Press files together, converting the Ears files generated by Pub and Bravo into Press format, selecting certain pages from a Press or Ears file, or adding extra fonts to a Press file. The general command format is illustrated in the following example:

```
pressedit foo.press ← a.press b.ears 2 5 c.press 3 to 7 9 meteor9/f
```

This means "make a Press file **foo.press** from all pages of **a.press**, pages 2 and 5 of the Ears file **b.ears**, and pages 3, 4, 5, 6, 7 and 9 of **c.press**; add font **meteor9** to the fonts defined in **foo.press**". The resulting file will be arranged in the same order as the component input files.

Examples:

To convert an Ears file **foo.ears** *to a file* **foo.press** *in Press format:*

```
pressedit foo.press ← foo.ears
```

To extract pages 3 and 17 from a Press file **long.press**, *and put them in* **short.press**:

```
pressedit short.press ← long.press 3 17
```

To extract pages 5 through 12 from **foo.ears**, *and put them in* **short.press**:

```
pressedit short.press ← foo.ears 5 to 12
```

To add fonts **logo24** *and* **helvetica14** *to* **a.press**:

```
pressedit a.press ← a.press logo24/f helvetica14/f
```

Here the arguments on the right hand side of the arrow may be given in any order.

To make a blank, one-page Press file containing all three faces of **Timesroman10**:

```
pressedit blanktimes.press ← timesroman10/f timesroman10i/f timesroman10b/f
```

To append to the end of **chap3.press** *all the Press files with names* **fig3-1.press, fig3-2.press, fig3-3.press** *etc:*

```
pressedit chap3.press ← chap3.press fig3-*.press
```

Caution: when you combine files with Pressedit, try not to use different sets of fonts, or the same fonts in different orders. This will result in proliferation of *font sets*, making the file more bulky and creating other minor sources of inefficiency.

RAMLOAD

RAMLOAD is a program that acts as a microcode loader, using the output of the microcode assembler Mu. Since there are now two types of microcode memory for the ALTO, some distinction must be made. Hereafer, ROM means some combination of roms on the ALTO control board, and add-on goodies which hang on the end of the control board like debuggers with 512 words of ram. RAM means the extra board with 1K of ram which plugs into a slot in the processor.

RAMLOAD gets its parameters from the command line and default values. If you do not specify a parameter, the default is used. In addition there are some global switches which do other useful things as explained below:
GLOBAL SWITCHES (of the form RAMLOAD/switchlist)

- /R compare the micro binary file against the contents of the RAM and display differences.
- /V compare the micro binary file against the contents of the ROM and display differences.
- /C compare the micro binary file against the contents of the constant memory and display differences.
- /T Test the RAM and extra R registers by writing random numbers and then reading them back displaying differences and addresses.
- /O Same as /T but do not test the R registers.
- /N Do not request Confirming <CR> for any operation.

LOCAL SWITCHES (of the form foo/switch)

- /F use foo as the name of the micro binary file. Default is "BINFILE."
- /M use foo as the name of the instruction memory in the micro binary file. Default is "INSTRUCTION".
- /C use foo as the name of the constant memory in the micro binary file. Default is "CONSTANT".
- /V foo is an octal number. Use it as the boot locus vector. Bit 15 corresponds to task 0 (emulator). 0 means run task in the RAM. Default is #177777 - keep all tasks in ROM.
- /A foo is an octal number, representing the base address of a 5 word area in the RAM which RAMLOAD can use for utility purposes. Default is the top 5 words (#1772). See warnings below about restrictions for specific operations.
- /S foo is an octal number interpreted as the beginning address of the emulator main loop (START for microcode hackers). Default is the current START address, #20.

Note that global switches /V, /C, and /T do the same things that ;V, ;C, and ;T do in DEBAL. RAMLOAD in effect does a ;L, and also sets the boot locus vector. The /R global switch was added because it was easy and people might want to see if the microcode got smashed after a fiasco.

When RAMLOAD is called, it will first display what it thinks it is supposed to do as governed by the switches and defaults, and wait for a confirming carriage return. When this is received, it will attempt to open the micro binary file. If this is unsuccessful, it will put out a message to that effect. Next, operations specified by global switches will be performed (If the micro binary file could not be opened, the only tests possible are /T and /O). If no global switches were set, the program will assume you wanted to load, and do so without waiting for confirmation. Loading is a three step operation in which the first step, setting the boot locus vector, does not require an open micro binary file. This allows a user to change the boot locus vector without reloading the RAM, by specifying a nonexsistant file name for the micro binary file. The program will report the value the vector is set to. Steps two and three, unsnarling the micro binary file and loading its contents, obviously require an open file and will cause RAMLOAD to bomb if there is none. When the loading operation is complete, the number of instructions loaded, and the highest address will be reported ala DEBAL. Next the program will ask if you want to boot, thus moving the tasks specified in the boot locus vector into the newly loaded microcode in the RAM. If you confirm, and if you have an Ethernet board, the machine will do a software initiated boot. If you do not have an Ethernet, the boot will be a NOP, and a FINISH is executed. Hitting the boot button after the program is finished will work for those hermits who do not have Ethernets.

The routine which reads the micro binary file expects the limited subset of block-types that DEBAL puts out. If it encounters an unusual block-type (3, 5, or 6), it will endeavor to do the right thing, and continue on. When it is finished reading, if any unusual types were encountered, it will list how many of each it read. If the microcode was assembled using DEBAL, this is cause for grave doubts about the correctness of the file, since DEBAL will not currently generate these types.

Where the 5 word utility area is specified can have profound (ie. potentially disasterous) effects on the machine's operation if you are currently running from the RAM. While it is possible to load the RAM while executing in it, this is living very dangerously. However, if you must, observe the following caveats:

* if constant memory is being checked, and you are executing out of the low 256 locations, you are dead.

* the 5 word utility area must be specified in a place you will not be executing from during the RAMLOAD program. RAMLOAD always saves any word in RAM it modifies for utility purposes, and restores it when it is done, but while in use, it can have an arbitrary value.

A number of things can cause fatal errors during execution. If one happens, an error message is written in the system display area, and the program is aborted.

SCAVENGER

A subsystem for checking and correcting disk packs is available as SCAVENGER. Invoke it with no parameters and it will give you an opportunity to (1) change disks and (2) prevent it from altering your disk seriously (see below).

The scavenger does the following:

1. Corrects header blocks, prompting for confirmation.
2. Corrects check sum errors, by re-writing whatever came in, prompting for confirmation.
3. Discovers all well-formed files and all free pages. Any disk page (except page 0) that is neither free nor part of a well-formed file is considered bad.
4. Makes the serial numbers of all well-formed files are distinct.
5. Corrects the system's notion of what pages are free.
6. Corrects the system's latest serial number.
7. Corrects the directory to contain precisely the well-formed files. If a directory entry points into a chain of bad pages it attempts to salvage the file. If need be a directory is created from scratch.
8. Links all bad, unsalvaged pages together as part of the file Garbage.$.
9. Describes all changes to the disk in the file ScavengerLog, even those which were not actually performed.
10. Corrects leader page information. Changes to leader pages should not cause alarm. The information there is used as a hint by various systems.

The data in bad pages is not changed so you can attempt to reconstruct a lost file by suitable operations on Garbage.$, consulting ScavengerLog to interpret its contents.

A hopelessly smashed disk may be put back in shape by the following:

1. Invoke scavenger on a good disk and answer yes to "Do you want to change disks?"
2. Replace the good disk with the bad one.
3. Answer yes to "Is the new disk ready?" when the yellow ready light comes on.
4. Answer yes to "May I alter your disk to correct errors?"
5. If FTP lives on your disk, the scavenger will offer to invoke it rather than returning to the executive. Once you are in FTP you can receive critcal files (like Executive.Run or SysFont.AI) or evacuate your disk by sending files elsewhere. If the scavenger does not offer FTP, it is not there and you will have to do some more disk suffling to retreive files; i.e. invoke FTP from a good disk and change disks after you are in.

You should take precautions to avoid losing vital files (such as QUICKing your disk to another disk pack prior to running SCAVENGER).

PARC information

The following, more or less independent, procedure can be used to recover vital files that might have been lost during scavenging.

1. Invoke FTP on a good disk.
2. At an early point in the dialogue replace the good disk with the bad one and wait for the yellow ready light to come on.
3. Retrieve the needed files from MAXC (Executive.Run and FTP are the minimum required, I think.)
4. Quit out of FTP.
5. Run the scavenger. It will correct the DiskDescriptor file which became inaccurate during this process.

Swat, a BCPL-oriented debugger

Swat is a debugger meant to be used with the Alto operating system. While many of its features are BCPL oriented, it can be used on any Alto program. This document describes version 27 of Swat, which is compatible with Operating System versions 16 and greater.

1. History

Swat was designed and built by Jim Morris and Alan Brown during the summer of 1973. Bob Sproull added the error file mechanism and parity error logging during 1976. Peter Deutsch rewrote the command processor and added the command file facility in early 1977. David Boggs renovated the program in late 1978 adding mulitple proceed break points and TeleSwat, and Ed Taft added the help facility. Everyone agrees that the human interface is awful. Each person who has worked on Swat has added several more obscure commands while they were at it.

2. How it works

Swat is an external debugger: with the exception of a small piece of 'resident' code in your address space, it lives in a separate space. When Swat is invoked, the resident saves your state on the file Swatee, and swaps in Swat. References to your memory from within Swat go to the Swatee file. When you tell Swat to proceed, it saves itself on the file Swat, swaps you (the Swatee) in and resumes you. Your state at the time Swat got control is displayed in a window at the bottom of the screen. "AC0", "PC", etc are built-in symbols with which you can manipulate it.

3. Invocation

Swat may be applied to any program running under the operating system after it has been installed (see Installation below). There are six ways of getting its attention:

(1) Hold down the <control> and <left-shift> keys and then press the <Swat> key.

(2) Have your program execute the op-code 77400B.

(3) Invoke the Resume/S command (see below).

(4) Boot the file Dumper.Boot, normally by booting with the "DU" keys depressed.

(5) Type <programName>/! to the Alto command processor.

(6) Call the function CallSwat. Up to 2 arguments will be printed as BCPL strings. Thus CallSwat("No more memory")

4. Commands

The command scanner has suffix action symbols, all of which are control characters (e.g. ↑C). "n" is any BCPL expression (see Expressions below), "$" is escape except where noted, "cr" means carriage return, "lf" means line-feed. You can abort whatever Swat is doing at any time and get back to the top level command scanner by pressing the <Swat> key.

4.1. Help facility

Most debuggers have a terse and obscure command syntax, and Swat is no different. In fact it's worse since it doesn't follow DDT conventions. Typing "?" prompts you for a command character which Swat looks up in the file "Swat.help". Responding "?" to its prompt gives you a small table of contents for the rest of the help file.

4.2. Displaying cells

address↑D	prints the contents of n in decimal
address↑I	prints the contents of n as two 8-bit bytes
address↑N	prints the contents of n as an instruction
address↑O	prints the contents of n in octal
address↑S	prints the contents of n as a pair of characters
address↑V	prints address in octal and decimal

The last cell printed is called the open cell. ↑O, ↑D, ↑I, ↑N, or ↑S alone re-prints the open cell in the appropriate format. If you wish to print out a number of cells, beginning with the open cell, say n$↑D, n$↑I, etc. The last cell printed becomes the open cell.

lf (↑J)	opens and prints the contents of the next cell (after the open one) in the same mode.
↑W	opens and prints the cell before the open cell.
↑A	opens and prints the cell pointed at by the open cell.
↑E	opens and prints the cell at the effective address of the open cell.
n↑=	searches from the open cell+1 for a cell whose contents is n or whose effective address is n. Prints and opens that cell. A search can take quite a while: abort by hitting <swat>.

The last cell that was opened by any command except LF or ↑W is called the last open cell. Often you are stepping through code, follow a pointer with ↑E or ↑A, look around, decide it's not interesting and wish to resume where you were before following the pointer. You can get back to last open cell plus or minus one by:

$lf ($↑J)	open and print last open cell+1.
$cr ($↑M)	open and print last open cell.
$↑W	open and print last open cell-1

4.3. Changing cells

The contents of the open cell (if there is one) may be changed by typing an expression for the new value followed by a cr, lf or ↑W. A$B followed by cr, lf or ↑W stores A lshift 8 + B into the open cell.

4.4. Running the program

↑P resumes the program, i.e. proceeds.

address↑G resumes the program at address, i.e. goes there.

<procName>$<e1>$...$<en>↑C calls the BCPL procedure "procName" with parameters <e1>,...,<en> (n<6). If you wish one of the arguments to be a BCPL-format string, merely enclose it in quotes. Thus OpenFile$"Com.Cm."↑C will return a stream on the file. AC2 is assumed to contain a legal stack frame pointer and 'procName' will allocate a new frame on top of it. Often AC2 is not valid (e.g., Swat interrupted the program in the middle of allocating a frame), and calling a procedure at this point may not work. Most of the time Swat can detect this and warn you.

↑U restores the user's screen. Hitting the <swat> key brings back Swat.

↑K forces the user program to abort, just as if you had typed <left-shift><swat> while it was running.

4.5. Break Points

A Break point can be referred to by its address or by the index assigned by Swat when the break point was set. When printing or deleting a breakpoint, Swat reaches out into the user's address space to check that the break is still there.

address↑B sets a break at address

↑B set a break at the open cell

0$address↑B deletes the break at address

proceedCnt$address↑B sets a multiple-proceed break point at address. The breakpoint will take effect when it has been hit proceedCnt times, and then it will be deleted. Passing through a multiple proceed break point without stopping takes about 200 us.

index$↑B deletes the break with index index

0$$↑B deletes all breaks

$$↑B prints all broken locations.

$↑P removes the current break and proceeds.

address$$↑P sets a one-shot break point at address and then proceeds. A one-shot break point is one that is removed after it is hit.

stackIndex$↑P sets a break at a BCPL return point in the stack somewhere and proceeds from the present break. The parameter n specifies the frame number, where the most recent (top) frame is 0. Thus if ↑T typed out 0:GOO+56 1:HAM+5, 1$↑P would set a break at HAM+6 and proceed.

4.6. Stack Study

See Chapter 10 of the BCPL manual and section 4.8 of the Operating System manual for the details of a BCPL stack.

↑T prints the current PC and all return addresses in the call stack (symbolically), until an

inconsistency in the stack (usually signaling its end) is encountered. After each return address is listed the parameters passed to the procedure that will be returned to. Thus, if you see an entry like "3: FindIt+45--(14 177777)", the procedure FindIt was called with arguments 14b and -1 (fine point: 14 and 177777 are the first two local variables in FindIt's frame, which FindIt could have modified before Swat was called, in which case they won't be the values passed at call time.)

n↑T prints n (or less) frames starting with the top frame on the stack.

index↑F prints the parameters of the nth latest stack frame and sets the pseudo symbol "$" (not escape) equal to the base of that frame. If ↑T displayed something like 0:FOO+3, 1:BLETCH+10,... Type 1↑F to see the parameters that were passed to BLETCH. $ is set to the base of BLETCH's frame (i.e., $ points at the frame's back link: the first local variable is in $+4.

4.7. Symbol table

↑Y prompts you for the name of a symbol file. Type the name of the subsystem that's running. If it can't find a file with the name you typed, Swat appends ".syms" to it and looks up the resulting file name before reporting failure. If BLDR created the file FOO.RUN it also created FOO.SYMS, which gives the locations of all the static names. Only statics can be used in Swat. There are permanent built-in symbols for the interesting page-1 and high memory locations, BCPL runtime routines, and the user's state variables (AC0-3, PC, etc.).

4.8. Save/Restore

See 'Resumable files' below for more details:

↑L prompts you for a file name on which it saves the current Swatee.

↑Q prompts you for a file name which it installs as the current Swatee.

4.9. The Spy Facility

The spy can be used to estimate where the time is going on a percentage basis. It samples the PC every 30-milliseconds.

(1) Type ↑X and Swat will display how much user memory it needs for the metering code and tables.

(2) Probe around to find a block of storage of the required size, and tell Swat by typing

n↑X

where n is the first word of the block.

(3) Proceed to run the program.

(4) Once Swat gets control again you can type

$↑X

to display the results and terminate the spying activity, or

$$↑X

to display the results so far and continue the spying.

4.10. Miscellaneous

$↑Y	Prompts for the name of a (text) file from which Swat commands should be taken. Reading will continue across "proceeds" from breakpoints, but will be aborted if Swat is invoked by the keyboard (<control><left-shift><swat>) or by the standard break-point trap (77400B).
$$↑Y	Puts Swat into TeleSwat server mode. The keyboard is ignored: to regain local control hit the <Swat> key. For more on TeleSwat see the sections on Address Spaces and TeleSwat.
n↑R	Prints the value of R or S register n. You must have a RAM for this to work.
$↑R	Prints all of the R and S registers.
$$↑Z	Repeats the message that was displayed when Swat was invoked. This is sometimes useful if an error message has scrolled away as a result of poking around.

4.11. Address Spaces

↑Z prompts for the target address space. Swat can treat any file created by OutLd, any bank of memory, and any host in the internet (with the host's cooperation) as the Swatee: the address space into which you peer with Swat. The syntax for address spaces is:

filename	this is 'Swatee' for normal debugging, but can be any file created by OutLd (sysOut files (↑L) are in this category), or Dumper.
Bank0	Swat itself.
Bank1...3	the extended memory banks. These are only legal on AltoII XMs. No check is made that a bank actually exists. If it doesn't, or if it hasn't been written into since the Alto was powered up, you are likely to get parity errors.
[host]	a host that implements the server half of the TeleSwat protocol (usually another Swat). [host] can be either a name: [Boggs], or an internet address: [3#241#]. The square brackets are required: this is how Swat decides that you mean a [host] rather than a file.

4.12. Examples

X↑O↑D	prints the value of X in octal, then decimal.
func+3↑N 1f 1f	prints instructions 3, 4, and 5 of func.
1↑O7	sets location 1 to 7.
label↑B	sets a break at label
7562↑B	sets a break at location 7562B
SQRT$16↑C	calls the (user) function SQRT (the returned value is printed)
label+3↑G	transfers to the third instruction after label.
0↑T	prints the PC
0↑F	prints the parameters of the most recent call
2↑F	prints the parameters of the third most recently called procedure; then

$↑O	prints the saved stack pointer (frame!0)
$+1↑0	prints the return address (frame!1)
$+6↑O	prints the first local (if the procedure has 2 parameters).

5. Expressions

Expressions are as in BCPL with the following exceptions

'	means exclusive OR
\	means REMAINDER
|	means LSHIFT for positive arguments, RSHIFT for negative
~	means NOT

A string of digits is interpreted as octal unless suffixed by a "."

$ (not escape) is the base of the last opened stack frame (see ↑F above). Initially it is the last frame.

↑<static name>, "↑" followed immediately by a static name, means use the address of the static, not its value, even if it is a procedure- or label-type static.

. is the last opened cell

PC is the address of the cell containing the user PC. This is the address at which Swat will resume Swatee when you say ↑P.

AC1,...,AC3 are the addresses of the user's accumulators.

CRY is the address of the user's carry bit.

INT = on = non zero if interrupts where on when the Swat trap happened.

No function calls in expressions.

No relational operators (e.g. EQ)

No conditional expressions

No lv operator (well...see ↑<static name> above)

5.1. Examples

.-1↑O	prints the cell before the currently open cell.
.+1↑O	is like line-feed.
AC1↑O6	sets AC1 to 6
PC↑O72 ↑P	is like 72↑G
PC↑O lf lf lf lf	prints the PC and the AC's

The conventions for expression evaluation are not truly BCPL-like. "F↑0" will print the first instruction of

F if BLDR thought it was a procedure or label, but print the contents of static cell F if BLDR thought it was a variable. If F started life as a variable, but had a procedure assigned to it you must call it by "@F↑C" instead of "F↑C".

6. Resumable Files

The file Swatee is a snapshot of a running program and can be saved for subseqent resumption or examination. You can create a copy of Swatee by using COPY or, if you are in Swat, typing ↑L and giving a file name. This copies Swatee to the named file and appends some information internal to Swat -- the current symbol table and break point data.

There are several ways to restart resumable files:

1) Press the boot button while holding down the keys for the file.

2) Type the command (it is interpreted by the Exec)

RESUME file

If "file" is omitted Swatee is assumed.

RESUME/S file

writes file onto Swatee and invokes Swat.

3) While in Swat, type ↑Q and give a file name. The file is copied onto Swatee and Swat's internal information is restored to whatever was saved by the ↑L command that created the file. If the file was created in some way other than ↑L, the internal information is reset to an empty state.

7. TeleSwat

Swat implements a simple Pup protocol, TeleSwat, by which it can treat a machine anywhere in the internet as the Swatee (with the consent and cooperation of the other machine). The Swatee is made receptive to control from the network by typing $$↑Y. The controlling Swat's attention is directed at it by specifying the Swatee's network address as the target virtual memory (see the ↑Z command). When you tell the Swatee to proceed (↑P, ↑G, ↑U), you loose control: your Swat starts probing the Swatee once per second, but if the Swatee never returns, you must get help from someone at the other end. Each time a packet is sent, the cursor is inverted to let you know something is happening. Executing the opcode 77412b is equivalent to CallSwat(string1 [], string2 []) followed by $$↑Y.

8. Desperation Debugging

If the resident is broken so you can't use <Left-Shift><Control><Swat> to get to Swat to see what went wrong, then you are desperate. Press the boot button while holding down the keys for the file Dumper.Boot (the OS and InstallSwat conspire to make this be "DU" normally). This writes the existing memory onto Swatee with the exception of page 0 which is lost (Dumper lands in page 0 when you boot it). Also the display word (420b) is cleared. Finally, Swat is invoked.

9. Error Message Printing

Swat contains some facilities to aid in printing error messages. Because the Swat resident is almost always present when a program is running, an error message can be printed by simulating a Swat "break," and letting the Swat program decipher the error specification and print a reasonable message.

If Swat is invoked by the 77403b trap instruction, the contents of AC0 are taken to be a pointer to a BCPL string for a file name; AC1 is a pointer to table [errCode%ClearBit; p1; p2; p3; p4....], where errCode (0 le errCode le 32000.) is an error code, the p's are "parameters," and ClearBit is either 100000b (clear the Swat screen before printing the message) or 0 (do not clear).

The intended use is with a BCPL procedure like:

```
let BravoError(code, p1, p2, nil, nil, nil) be
  [
  code = code%UserClearScreenBit
  (table [ 77403B; 1401B ])("bravo.errors", lv code)
  // do a "finish" here if fatal error
  ]
```

The error messages file is a sequence of error messages, searched in a dumb fashion. An error message is:

a. An unsigned decimal error number (digits only)
b. Followed optionally by:
   C Always clear the screen before printing the message
   M (see below)
   L Log the error via the Ethernet.
c. Followed by a <space>.
d. Followed by text for the message, including carriage returns, etc.
   If you wish to refer to a parameter, give:
   $
   followed by a digit to specify the parameter number (1,2,....)
   followed by a character to say how to print the parameter:
      O = octal
      D = decimal
      S = string (parameter is pointer to BCPL string)
   (example: $1D will print parameter 1 in decimal)
   The quote character is <escape>.
e. Followed by $$.

After the message is typed, if M was specified, the message "Type <control>K to kill, or <control>P to proceed." is typed out.

10. Parity Error Information

When the Alto detects a parity error, Swat is usually invoked to print a message about the details of the error. It then attempts to "log" the error with an Ethernet server responsible for keeping maintenance information. If the server is not operating, or if your Alto is not connected to an Ethernet with such a server, simply strike the <Swat> key, and the familiar "#" will appear.

In many cases, you will want to continue execution of your program after a parity error is detected. Simply type <control>P to Swat.

11. Installation

Get the file InstallSwat.Run. Then invoke it to create Swat (the debugger), Swatee (the swap file for the user's memory image), and Dumper.Boot (the desperation debugger invoker). InstallSwat.Run may be deleted after it has been run once. Use the Exec's BootKeys command to discover the keys to depress for Dumper.Boot; normally they are "DU".

InstallSwat.run is the Swat program. When invoked it, it hooks up to the current operating system, initializes itself, and then OutLds all of core including the OS (suitably Junted and slightly patched) onto the file Swat.

12. Caveats

1. Swat has about 1k of resident code in high memory. This code is not changed when new subsystems come in. Therefore re-boot if it seems to be in a bad state. Swat can get itself into a bad state too. SYSINing (↑Q) Swatee is a very effective general purgative; ignore the warning message - its doing exactly what you want it to. If all else fails, make sure you have a clean copy of the OS, and then reinstall Swat by running InstallSwat.run.

2. Instructions 77400B - 77777B are used by Swat. The actions of some of these (e.g. 77401B) are published; you get what you deserve if you use the unpublished ones. Location 567B (in the trap vector) is used.

3. Interrupt channel 8 (00400B) is used by the resident for keyboard interrupts (getting to swat via a <control><left-shift><swat> key combination).

4. A program fetching data from a broken location will get 774xxB.

5. While most interrupt routines are reasonably polite and always resume the interrupted code where it left off, the politeness of Swat's keyboard interrupt is entirely in the hands of the person at the controls. If he re-starts by saying ↑P, all goes well; but he may say ↑G or ↑C. Therefore

   a) You should disable the keyboard interrupt by anding 77377B into 453B during critical sections of code (once they are debugged).

   b) Expect occasional anomalies after ↑C or ↑G is used.

6. The mappings between symbols and addresses are naive about BCPL's block structure.

   a) If a symbol is defined twice or more you get the lowest address.

   b) An address is mapped into a procedure name plus a displacement for symbolic type out (e.g. for ↑T). If procedure A is defined inside procedure B, most of B's addresses will be typed as if they were A's.

7. If a disk error prevents swapping, the offending disk control block and label are displayed in the "boot-lights" manner.

8. Locations 700b through 707b are used to save the machine state before each swap.

9. If a file created on a different disk is resumed by booting, invoking Swat may not work because Swat and Swatee may not reside at the same disk addresses on the different disks. This difficulty does not occur if the Exec's RESUME command is used, since it will fix up the addresses before invoking it.

Software and Utilities for Trident Disks: Tfs and Tfu

1. Introduction

This document describes Bcpl-based software for operating any of the family of Trident disk drives attached to an Alto using a "Trident controller card" (the software presently deals with the T-80 and T-300 models). Hardware and diagnostic information can be found in the document "Trident disk for the Alto" (on <ALTODOCS>TRIDENT.EARS), by Roger Bates.

The software documentation is divided into three parts: (1) a brief "how-to" section describing the software package available for operating the Trident; (2) a section describing the utility program Tfu; and (3) a section describing the software package in more detail. There is a short revision history at the end. (Documentation for the Triex program, formerly included here, has been eliminated. Triex is now needed only for hardware checkout and is not required during normal operation.)

The Tfs package and utilities all assume that the disk is to be formatted with 9 sectors per track, 1024 data words per sector. Thus a T-80 disk has a capacity (815 tracks, 5 surfaces, 9 sectors, 1024 words per sector) of 36,675 pages or 37,555,200 words. A T-300 (19 surfaces rather than 5) has a capacity of 139,365 pages or 142,709,760 words; however, due to the restriction of virtual disk addresses to 16 bits, a single file system may utilize only about 47 percent of this capacity, and it is necessary to construct multiple file systems in order to make use of the entire disk.

Because of bandwidth limitations, it is unwise to operate the Trident disk while the Alto display is on. Although the Tfs package will save the display state, turn it off, run the disk, and restore the display for every transfer, the user may prefer to turn the display off himself. The Tfs management of the display causes the screen to flash objectionably whenever frequent calls to Tfs are underway.

The present version of the software conforms to the new Alto time standard and runs only under Operating System version 14 or newer.

2. Trident File System (Tfs) software package

The software for operating the Trident disk is contained in <Alto>Tfs.Dm, and consists of the following relocatable files: TfsInit.Br, TfsBase.Br, TfsA.Br, TfsWrite.Br TfsCreate.Br, TfsClose.Br, TfsDDMgr.Br, TfsNewDisk.Br, TfsSwat.Br, and TriConMc.Br. The definitions file Tfs.D is also included.

Included also are the Trident microcode source files, TriConMc.Mu and TriConBody.Mu. These are needed if you want to load other microcode into the Ram along with the Trident microcode.

The LoadRam.Br file, formerly included as part of the Tfs, is now available as a separate package.

2.1. Initializing the microcode

Operating the Trident requires special microcode that must be loaded into the RAM before disk activity can start. The procedure LoadRam will load the RAM from a table loaded into your program (it is actually part of TriConMc.Br). It will then "boot" the Alto in order to start the appropriate micro-tasks in the RAM. (This booting process is "silent" -- it does not re-load Alto memory from the file Sys.Boot, but instead lets your program continue.) The standard way to call LoadRam to load the Trident disk microcode is:

```
external DiskRamImage
external LoadRam

let result=LoadRam(DiskRamImage, true) //Load and boot
if result ls 0 then
    [
    Ws("The Alto has no RAM or Ethernet board.")
    Ws(" Cannot operate Trident")
    finish
    ]
```

After LoadRam has returned successfully, the code of LoadRam and TriConMc may be overlaid with data -- they are no longer needed.

When exiting a program that has micro-tasks active in the RAM, it is helpful to "silently" boot the Alto so that all micro-tasks are returned to the ROM. If this is not done, subsequent use of the RAM may cause some running micro-task to run awry. To achieve the "silent boot," simply call the procedure TFSSilentBoot() at 'finish' time or as part of a 'user finish procedure'.

For further information, consult the LoadRam package documentation.

2.2. Initializing the Trident drive

Once the RAM has been loaded, the Trident disk can be initialized. The procedure TFSInit will do this, provided that a legal file structure has previously been established on the drive (see Tfu Erase, below). The procedure returns a "disk object," a handle which can be used to invoke all the disk routines. This disk object (or "disk" for short) can be passed to various Alto Operating System procedures in order to open streams on Trident disk files, delete Trident disk files, etc.

tridentDisk = TFSInit(zone, allocate [false], driveNumber [0], ddMgr [0], freshDisk [false])

zone — You must provide a free-storage pool from which memory for the disk object and possibly for a buffer window on the disk bit table can be seized. The zone must obey the normal conventions (see Alto Operating System Manual); zones created by InitializeZone are fine.

allocate — This flag is true if you wish the machinery for allocating or de-allocating disk space enabled. If it is enabled, a small DDMgr object and a 1024-word buffer will be extracted from the zone in order to buffer the bit table (unless you supply a ddMgr argument, described below).

driveNumber — This argument, which defaults to 0, specifies the number of the Trident disk drive being initialized. If the drive is a T-300, the left-hand byte specifies the number of the file system to be accessed on that drive, in the range 0 to 2. (For further information, consult the section entitled 'Disk Format'.)

ddMgr — This argument, which defaults to 0, supplies a handle on a 'DiskDescriptor Manager' (DDMgr) object, whose responsibility it is to manage pages of the DiskDescriptor (bit table), which, on the Trident, must be paged into and out of memory due to its considerable size. If this argument is defaulted, a separate DDMgr will be created upon each call to TFSInit, at a cost of a little over 1024 words. If you intend to have multiple Trident drives open simultaneously, you may conserve memory by first issuing the call 'ddMgr = TFSCreateDDMgr(zone)' and then passing the returned pointer as the ddMgr argument in each call to TFSInit, thereby permitting the single ddMgr to be shared among all drives. (This argument is ignored unless the allocate argument is true.)

freshDisk — Normally, TFSInit attempts to open and read in the DiskDescriptor file in order to obtain information about the file system. However, if freshDisk is true, this operation is inhibited and the corresponding portions of the disk object are set up with default values. This operation is essential for creating a virgin file system.

tridentDisk The procedure returns a disk object, or 0 if the Trident cannot be operated for some reason. The most likely reasons are:

1. No Trident disk controller plugged into the Alto.

2. No such disk unit, or disk unit not on-line.

3. Can't find SysDir, can't open DiskDescriptor, or DiskDescriptor format is incompatible. (These errors can't happen if freshDisk is true.)

Important: If the AC power to drive 0 is turned off or no drive 0 is connected, it is not possible to operate any drive. (Drive 0 need not be on-line, however.) This is due to a hardware bug that has been deemed too difficult to fix.

After TFSInit has been executed, the code can be overlaid, as it is not used for normal disk operation.

2.3. Closing the Trident disk

When all operations on the disk are completed, the TFSClose procedure will insure that any important state saved in Alto memory is correctly written on the disk. This step can be omitted if the 'allocate' argument to TFSInit was false (assuming you don't mind the loss of the storage that was extracted from 'zone' by TFSInit).

```
TFSClose(tridentDisk, dontFree [false])
```

The second argument is optional (default=false), and if true will not permit the DiskDescriptor Manager (DDMgr) to be destroyed. This option is useful in conjunction with the 'ddMgr' argument to TFSInit.

2.4. Example

Following is an example that uses the Trident disk system and demonstrates the procedures described above. Note that the calls on operating system disk stream routines all pass a private zone to use for stream structures, rather than the default sysZone. The reason is that streams on Trident disks require large buffers (1024 words) which quickly exhaust the available space in sysZone. In addition, the stream routines will consume more stack space when operating the Trident disk than they do when operating the standard Alto disk.

Since the Alto OS does not know about Trident disks, a call to Swat will not properly wait for all Trident transfers to complete, with consequent undefined results. This problem is easily remedied through use of an assembly-language Swat context-switching procedure TFSSwat, which is included as part of the TFS package. The example shows how it is set up.

```
//Example.bcpl -- TFS Example
//Bldr Example TfsBase TfsA TfsWrite TfsCreate TfsClose TfsDDMgr
//  TfsSwat TfsInit LoadRam TriConMc

get "streams.d"

external [
    TFSInit
    TFSClose
    TFSSilentBoot
    LoadRam
    DiskRamImage

    OpenFile
    Closes
    Puts
```

```
    DelcteFilc

    InitializeZone
    SetEndCode
    TFSSwatContextProc
    lvUserFinishProc
    lvSwatContextProc
    ]

static [ savedUFP; savedSCP; TFSdisk = 0 ]

let TryIt() be
[
    let driveNumber=0
    let zonevec= vec 3000
    let TFSzone = InitializeZone(zonevec, 3000)

//Initialize the RAM:
    let res=LoadRam(DiskRamImage, true)
    if res ls 0 then [ Ws("Cannot load the RAM."); finish ]

//Set up to cleanly finish or call swat
    savedUFP = @lvUserFinishProc
    @lvUserFinishProc = MyFinish
    savedSCP = @lvSwatContextProc
    @lvSwatContextProc = TFSSwatContextProc

//Initialize the disk:
    TFSdisk = TFSInit(TFSzone, true, driveNumber)
    if TFSdisk eq 0 then
        [ Ws("Cannot operate Trident disk"); finish ]

//Reclaim space used by initialization code:
    SetEndCode(TFSInit)    //Overlay TFSinit, LoadRam, TriConMc

//Now we are ready to operate the disk:
    DeleteFile("Old.Bad", 0, 0, TFSzone, 0, TFSdisk)

    let s=OpenFile("New.Good", ksTypeReadWrite, 0,0,0,0,
                TFSzone, 0, TFSdisk)

    for i=1 to 1000 do
     for j=1 to 1000 do Puts(s, $a) //Write a million bytes!

    Closes(s)

    finish
]

and MyFinish() be
[
    if TFSdisk ne 0 then TFSClose(TFSdisk)
    @lvUserFinishProc = savedUFP
    @lvSwatContextProc = savedSCP
    TFSSilentBoot()
]
```

3. Trident File Utility, Tfu

The Tfu utility (saved on <Alto>Tfu.Run) is used to certify a new Trident pack for operation, to initialize a pack with a virgin file system, and to perform various file copying, deleting, and directory listing operations. Commands are given to Tfu on the command line: immediately following the word "Tfu" is a sub-command name (only enough characters of a sub-command are needed in order to distinguish it from other sub-commands), followed by optional arguments. Several subcommands may appear on one command line, separated by vertical bars. Thus "TFU Drive 1 | Erase" will erase drive 1. There must be a space on each side of the vertical bar.

In what follows, an "Xfile" argument is a filename, perhaps preceded by a string that specifies which disk is to be used:

```
s:name.extension     -- use standard Alto system disk
tn:name.extension    -- use Trident drive n (n=0 to 7)
name.extension       -- use default disk (Trident)
```

The "default disk" is always a Trident drive; the identity of the drive is set with the Drive command.

TFU DRIVE driveNumber

> This command sets the default Trident drive number to use for the remainder of the command line. The default drive is effectively an 'argument' to the CERTIFY, ERASE, DIRECTORY, CONVERT, and BADSPOTS commands. (On a T-300, file systems 0, 1, and 2 are specified as 'x', '40x', and '100x', where 'x' is the actual unit number.)

TFU CERTIFY [passes]

> This command initializes the headers on a virgin Trident disk pack, then runs the specified number of passes (default 10) over the entire pack, testing it using random data. Any sector exhibiting an uncorrectable ECC error, or correctable ECC errors on two or more separate occasions, is permanently marked unusable in the pack's bad page list. This information will survive across all subsequent normal file system operations (including TFU ERASE), but may be clobbered by the Trlcx program.
>
> This command should be executed on every new Trident pack before performing any other operations (such as TFU ERASE). 10 passes of TFU CERTIFY are adequate for reasonably thorough testing, though more are recommended for packs to be used in applications requiring high reliability. The running time for TFU CERTIFY is approximately 3 minutes per pass on a T-80 and 9 minutes per pass on a T-300.
>
> TFU CERTIFY may be terminated prematurely by striking any character to get its attention, then typing 'Q'. Subsequent runs of TFU CERTIFY will not clobber the existing bad page information but rather will append to it. It is recommended (though not necessary) that TFU CERTIFY be executed before each TFU ERASE so as to pick up any new bad spots that may have developed.
>
> TFU CERTIFY ordinarily asks you to confirm wiping out the disk before going ahead and doing so; however, the /N global switch may be used to indicate that no confirmation is necessary.

TFU BADSPOTS

> Displays the addresses of all known bad spots on the disk pack mounted on the default drive.

TFU RESETBADSPOTS

> Resets the bad spot table of the disk pack mounted on the default drive. (Note that TFU CERTIFY appends to the existing bad spot table.) There should normally be no need to execute this command, but it may be useful, for example, after a disk pack is cleaned, if the known bad spots were caused by dirt.

TFU ERASE [tracks]

This command initializes (or reinitializes) a file system on the pack mounted on the default Trident drive, after asking you to confirm your destructive intentions (overridden by the /N global switch). The tracks argument specifies how many tracks of the drive are to be included in the file system; it defaults to the maximum possible. If smaller numbers are used, the initialization is correspondingly faster. In any case, tracks beyond the one specified are available for use outside the confines of the file system. (Note that one "track" is 45 pages; this corresponds to one cylinder on a T-80 and to nothing in particular on a T-300.)

The disk pack should previously have been initialized and tested by means of the TFU CERTIFY command.

The DiskDescriptor file is normally located in the middle of the file system so as to minimize average head movement between DiskDescriptor and file pages. However, this does limit the maximum size contiguous file that can be created to a little less than half the file system. If you wish to create a contiguous file larger than that, use the /B local switch (i.e., TFU ERASE/B) to force the DiskDescriptor to be located at the beginning of the file system instead.

TFU COPY Xfile ← Xfile

This command copies a file in the direction of the arrow. The destination file may be optionally followed by the switch /C, in which case (provided it is a Trident disk file), the file will be allocated on the disk at consecutive disk addresses. (Note: More precisely, an attempt will be made to perform such an allocation. If the attempt fails, you will sometimes get an error message. The best way to verify that a file is contiguous is to use the "address" command, below.)

TFU CREATEFILE Xfile pages

This command creates a contiguous file named Xfile with length "pages."

TFU DELETE Xfile

This command deletes the given file.

TFU DIRECTORY [Xfile]

This command lists the directory of the default Trident drive on the file Xfile; if Xfile is omitted, each entry will be typed on the display. When the display fills up or the listing is finished, Tfu waits for you to type any character before proceeding. A somewhat more verbose listing can be achieved with TFU DIR/V.

TFU ADDRESS Xfile

This command reads the entire file and prints a list (in octal) of virtual disk addresses of the file pages. Typing any character will proceed to the next output line.

TFU CONVERT

An incompatible change in the format of DiskDescriptor was made in the Tfs release of July 24, 1977. The current Tfs software will refuse to access Trident disks written in the old format (specifically, TFSInit will return zero). The TFU CONVERT command reformats the DiskDescriptor to conform to current conventions (it is a no-op if applied to a disk that has already been converted). Once you have converted all your Trident disks, you should take care to get rid of all programs loaded with the old Tfs, since the old Tfs did NOT check for version compatibility.

TFU EXERCISE passes drive drive drive ...

This command embarks on a lengthy "exercise" procedure; it is repeated 'passes' times

(default = 10), and uses the disk drives listed after 'passes' (if none are specified, all drives that are on-line are used). It operates by making a series of files (test.001, test.002 etc.) on the disk packs, and performing various copying, deleting, writing and positioning operations. The files are deleted when the exercise finishes. It is not essential that the packs be fully erased initially; the procedure for building test files will try to fill up the disk, just short of overflowing. The test takes 20 to 30 minutes per full pack per pass.

One or more of the following global switches may be specified (i.e., a command of the form TFU/switch EXER...):

/W Use a systematic data pattern when writing files, rather than arbitrary garbage.

/C Carefully check the data read from the disk (implies /W). Use of this switch makes the test run considerably slower than normal.

/D Leave the display on during Trident disk transfers. This causes data late errors to occur and thereby exercises the error recovery logic.

/E Turn the Ethernet on during Trident disk transfers, with results similar to /D.

4. The Tfs software package in more detail

If programmers wish to interface the the Trident disk at levels lower than Operating System streams, the Tfs package provides an additional interface. The "disk" object created by TFSInit has a number of abstract operations defined on it, which the Tfs package implements. Documentation for these operations can be found in the Alto Operating System Manual in the section labeled "Disks and Bfs." The catalog of available procedures is:

```
In TfsBase.Br and TfsA.Br:
    ActOnDiskPages(disk, CAs, DAs, ....)
    RealDiskDA(disk, vda, ....)
    VirtualDiskDA(disk, ....)
    InitializeDiskCBZ(disk, cbz, ...)
    DoDiskCommand(disk, cb, ...)
    GetDiskCb(disk, cbz, ...)

In TfsWrite.Br:
    WriteDiskPages(disk, CAs, DAs, ....)
    AssignDiskPage(disk, vda)*

In TfsCreate.Br
    CreateDiskFile(disk, name, ....)*
    DeleteDiskPages(disk, CA, ....)*
    ReleaseDiskPage(disk, vda)*

In TfsClose.Br
    CloseDisk(disk, dontFree)
```

The items with *'s following may be invoked only if the disk object was created with the 'allocate' argument set to true. WriteDiskPages may be invoked even if 'allocate' is false, provided it never allocates new disk space. It should be noted that the standard Alto Streams package invokes WriteDiskPages even for files opened for reading only, and that TFSInit uses Streams to read in the DiskDescriptor. Hence it is necessary that all of the Tfs modules (TfsBase, TfsA, TfsWrite, TfsCreate, and TfsDDMgr) be loaded in order to avoid undefined 'external' references. However, after initialization is complete, the space occupied by TfsCreate and TfsDDMgr may be reclaimed if you do not intend to allocate or delete pages,

and TfsWrite may be discarded if you are not using streams but rather are calling ActOnDiskPages directly.

The TfsWrite and TfsCreate modules require that TfsDDMgr.Br (or some equivalent) be loaded. This module provides the standard primitives necessary for managing the DiskDescriptor. The DDMgr is an 'object', so it may be replaced by one of your own devising so long as it provides equivalent operations. An example of this would be to manage pages of the DiskDescriptor as part of a more general virtual memory mechanism (perhaps through use of the Alto VMem package). A complete description of the required DDMgr operations may be found as comments at the beginning of TfsDDMgr.Bcpl.

In addition to the standard "actions" defined in Disks.d, Tfs permits the following. These actions are defined in Tfs.d and are available only on Trident disks.

DCreadLnD	Read header, read label, no data.
DCreadnD	Check header, check label, no data.
DCwriteLnD	Check header, write label, no data.

These actions neither read nor write the data record and therefore do not require a buffer to be provided.

CreateDiskFile has a special feature for operating the Trident disks -- an optional seventh argument. If this argument (pageBuf) is present, it is assumed to point to a 1024-word buffer that will be used to create the leader page for the file. This feature may be used to save stack space in CreateDisk file and/or to write interesting data into the portion of the leader page not used by the file system (only the first 256 words are used by the file system; the remainder has no standard interpretation).

VirtualDiskDA returns fillInDA as the virtual address for a real disk address that is either illegal or outside the confines of the file system.

The procedures for creating and destroying the disk object, TFSInit and TFSClose, were explained above. The procedure TFSWriteDiskDescriptor(disk) will write out onto the disk all vital information about the disk that is presently saved in memory. If you write programs that run the disk for extremely long periods of time, it is wise to write the disk descriptor occasionally. The only automatic call on TFSWriteDiskDescriptor is performed by TFSClose.

TfsInit.Br contains a procedure TFSDiskModel(disk) that returns the model number (80 or 300) of the drive referenced by the disk handle. This is useful in deciding whether to open a second or third file system on a T-300.

A lower level of access is permitted with the routines InitializeDiskCBZ, GetDiskCb, and DoDiskCommand, analogous to the Bfs routines described in the Operating System Manual. Users of these routines may wish to retrieve source files for the Tfs package and examine the definitions in Tfs.D and the actual disk operation in some detail. Sources are on <AltoSource>TfsSources.Dm.

4.1. TFSNewDisk

The TFSNewDisk procedure, defined in TfsNewDisk.Br, "erases" a disk (formatting it and making all its pages appear free) and creates a virgin Alto file system (SysDir and DiskDescriptor). It is called by:

success = TFSNewDisk(zone, driveNumber [0], diskSize [default], ddVDA [diskSize/2])

The zone passed to TFSNewDisk must be capable of supplying about 3500 words of storage. If the drive is a T-300, the driveNumber may include a file system number (0 to 2) in its left byte, as is the case for TFSInit. The diskSize argument is the number of disk pages to be included in the file system; it defaults to the maximum possible, which is all of a T-80 or a little less than half of a T-300. ddVDA is the virtual disk address at which to locate the DiskDescriptor file; see the TFU ERASE command for elaboration on this.

TFSNewDisk returns true if successful.

4.2. DiskFindHole

The procedure DiskFindHole, in DiskFindHole.Br, can be used to locate a "hole" of available space in the disk bit table. The call:

```
virtualDA = DiskFindHole(disk, nPages)
```

will attempt to locate a contiguous hole nPages long. If it fails, the procedure returns -1, otherwise the virtual disk address of the first page of the hole.

In order to create a contiguous file, it is first necessary to create the minimal file with a leader page at the given disk address and then to use Operating System or Tfs routines to extend the file properly. The first step is achieved by calling

```
ReleaseDiskPage(disk, AssignDiskPage(disk, vda-1))
```

where 'vda' is the desired disk address (i.e., the result returned by DiskFindHole). This value will control the selection of an initial disk address for the leader page. Once the file is created, it is wise to extend it to its final length immediately, as other disk allocations might encroach on the "hole" that was located.

For example, if we are using the Operating System, we might proceed as follows:

```
let nPages=433        //Number of data pages needed.
let vda=DiskFindHole(disk, nPages+2)
            //(+2= 1 for leader, 1 for last page)
test vda eq -1
    ifso Ws("Cannot find a hole big enough") ]
    ifnot ReleaseDiskPage(disk, AssignDiskPage(disk, vda-1))

let s=OpenFile("New.Contiguous",ksTypeWriteOnly,0,verNew,0,0,0,
        TFSzone, 0, disk)
PositionPage(s, nPages) //Make the file the right length
Closes(s)
```

5. File structure on the Trident disk

The file structure built on the Trident disk by Tfs (Trident File System) is as exact a copy of the Alto file structure built Bfs (Basic File System) as is possible. Certain exceptions are present due to hardware and microcode differences. The Alto Operating System Reference Manual should be consulted for all file formats and internal information not presented here.

5.1. Disk Format

The Trident disk unit and pack, as it comes from Calcomp, is set up to run with the following parameters:

number of cylinders: 815
number of surfaces: 5 (T-80), 19 (T-300)

TFU CERTIFY will format each surface in the standard Tfs format:

number of sectors per track: 9
header words per sector: 2
label words per sector: 10
data words per sector: 1024

Thus, a T-80 disk will have 9*5*815 = 36,675 sectors = 37,555,200 words. Sector 0 will not be used by Tfs. All but sector 0 will be available to the file system.

Ordinarily, Tfs utilizes only the first 383 cylinders (= 65,493 sectors = 67,064,032 words) of a T-300 disk. This is the largest integral number of cylinders that can be addressed using a 16-bit virtual disk address. The 16-bit virtual address limitation is deeply embedded in all existing higher-level Alto file system software, so changing the Tfs interface to permit a larger virtual address space would be impractical.

Instead, Tfs permits one to obtain another, entirely independent disk object for referencing the second 383 cylinders of the same T-300, thereby permitting a separate, self-contained file system to be constructed. This is done by passing a '1' in the left byte of the 'driveNumber' argument to TFSInit or TFSNewDisk (that is, drive '#400' refers to the second file system on a T-300 pack mounted on drive 0). A third file system (number '2', drive '#1000') may also be constructed, but it contains only 49 cylinders (= 8379 pages, only 6 percent of the disk's total capacity), so doing so is probably not worthwhile.

5.2. Disk Header and Label

On the Trident, a real disk address requires two words to express, rather than the single word on the Diablo 31. Also, microcode considerations gave rise to a reordering of the entries in the Label. The result is that both the header and label formats are different for the Trident. The Trident format follows. If you are interested in this level of detail, the file Tfs.d (contained within <Alto>Tfs.dm) should be consulted.

```
// disk header
structure DH:
    [
    track word
    head byte
    sector byte
    ]

// disk label
structure DL:
    [
    fileid word lFID
    packID word
    numChars word
    pageNumber word
    previous @DH
    next @DH
    ]
    manifest lDL = size DL/16
```

5.3. Disk Descriptor

Every valid Tfs disk has on it two files which must contain the state information necessary to maintain the integrity of the file system. The Tfs system directory, "SysDir.", is identical in format and purpose with its Bfs counterpart. However the Tfs disk descriptor file, "DiskDescriptor.", while identical in purpose, is formatted differently to allow easy manipulation of the bit table (which, for the Trident, has to be paged in and out of memory). This difference in format should not be evident to even low-level Trident users (unless you write your own DDMgr), but is mentioned here for completeness.

5.4. Bad Page Table

Tfs and Tfu observe the standard Alto file system convention of recording -2's in the labels of all known bad pages. However, if this were the only location of such information, "erasing" a disk (to create a virgin file system) would require two passes over the entire disk: one to collect the addresses of all known bad

pages and one to mark all remaining pages deleted. This would require an excessive amount of time, particularly on a T-300.

A duplicate table of known bad pages is therefore recorded on physical page zero (= cylinder 0, head 0, sector 0) of the disk. This page is not available to the file system for other reasons having to do with end-of-file detection. The format of the table is given by the BPL structure, which is defined in Tfs.d. Note that the entries are REAL disk addresses and can therefore refer to any page on the disk regardless of whether or not such a page is accessible through the file system. (A T-300 has only one bad page table, even if it contains several file systems.)

The TFU CERTIFY command is responsible for testing the pack and building the bad page table. The TFSNewDisk procedure (called by TFU ERASE) is careful not to clobber this information but rather to propagate it to the other places where it is needed (namely, the disk bit table and the labels of the bad pages themselves). As a result, the bad page information, once initialized, will survive across all normal operations on the disk, including "erase" operations.

There does not presently exist any facility for manually appending to this list when new bad pages are discovered. Experience to date with the Trident disks (which provide correction for error bursts of up to 11 bits in length) has shown that such a facility is probably not needed. Thorough testing of disks (using TFU CERTIFY) is recommended before putting them into regular use, however.

6. Revision History

July 24, 1977

Incompatibilities:

The format of DiskDescriptor has changed. The new Tfs cannot access old disks or vice versa. See description under "TFU CONVERT".

There is now another file, TfsA.Br, that is logically part of TfsBase.Br and must be loaded along with it. It contains assembly-language code formerly included as "tables" in TfsBase.Br.

New Features:

Partial support for T-300 disks.

Conforms to new conventions for maintaining addresses of known bad pages.

TFSInit checks for valid SysDir leader page and DiskDescriptor version.

Count of bit table discrepancies added to DiskDescriptor. (These are pages falsely claimed to be free in the bit table.)

VirtualDiskDA returns fillInDA for illegal real disk addresses.

Additional Trident-specific disk actions.

Tfs is now entirely reentrant, so it is safe for the Idle() procedure to give control to another process that in turn calls Tfs procedures.

October 21, 1977

Incompatibilities:

The former TfsWrite module has been broken into four pieces: TfsWrite, TfsCreate, TfsClose, and TfsDDMgr. In most applications, all four must be loaded.

The 'sharedBT' argument to TFSInit has been replaced by a 'ddMgr' argument. The mechanism for sharing a bit table buffer among multiple drives has been entirely changed. (Programs that omit this argument are unaffected by the change.)

The TFSCreateVDA static has been removed. In its place is a new procedure TFSSetStartingVDA(disk, vda) that serves the same purpose.

The syntax of the TFU EXERCISE command has been changed. It is now 'TFU EXERCISE <passes> <list of drives>', and <list of drives> defaults to all drives that are on-line.

New features:

Complete support for T-300 disks. In conjunction with this, the TFSDiskModel procedure has been added.

It is now possible for DiskDescriptor pages to be managed externally (perhaps through some sort of virtual memory mechanism) by use of a user-defined 'DiskDescriptor Manager' object.

TFSSilentBoot procedure added.

November 9, 1977

Incompatibilities: None.

New features:

TFU CERTIFY and TFU BADSPOTS commands added. TFU CERTIFY initializes the headers on a virgin disk pack and then runs repeated tests over the entire pack, permanently recording any bad spots that it finds. This command replaces all the normal uses of the Triex program, documentation for which has been removed.

Microcode modified for more efficient reading on Alto-IIs (by about 25%).

February 26, 1978

Incompatibilities: Software updated to new time standard; will not run under OS versions earlier than 14.

New features: Microcode source now in two parts, to facilitate combining it with other microprograms.

December 15, 1978

Incompatibilities: some of the TFS DDMgr procedures renamed (used internally).

New features: returnIfNoCb argument to TFSGetCb; ddVDA argument to TFSNewDisk; TFU ERASE/B option to maximize contiguous free storage; TFU RESETBADSPOTS command added; TFS and TFU should run on Dorado.

June 25, 1979

Incompatibilities: none.

Changes: Optional "hintLastPage" argument added to ActOnDiskPages, WriteDiskPages, and DeleteDiskPages; several minor bugs fixed.

July 17, 1979

Incompatibilities: The structure of a DSK (and therefore a TFSDSK) changed, so programs that get "Tfs.d" must be recompiled; TFSSetStartingVDA(disk, vda) procedure removed--instead use ReleaseDiskPage(disk, AssignDiskPage(disk, vda-1)).

Changes: New operations InitializeDiskCBZ, DoDiskCommand, and GetDiskCb added to the DSK object in preparation for OS 17. Note that the new TFS will work under earlier versions of the OS, but the old TFS will not work under OS 17.

Note: Pending release of OS 17, the OS 17 version of Disks.d is released as part of the TFS package. This is required to compile programs that get "Tfs.d" or that use the new DSK generic operations.

ViewData -- 2D projections of 3D data on Display Screen

ViewData is a BCPL subsystem that will draw a picture of a file of data on your dispaly screen, and allow you to interactively control your point of view on the data. It handles only a two-dimensional array of single-word values (i.e. a three-dimensional surface, a function of two variables evaluated over a regular finite grid). Here is a list of features:

1) ViewData accepts input in the simplest possible file format: an optional header of any number of words (with any contents, which are ignored), followed by a block of (signed) data words of any size, with any dimensions.

2) ViewData takes all parameters from a dialog with the user via keyboard and mouse. By specifying different header sizes and dimension sizes, the user can exercise limited control over the selection of data from his file.

3) ViewData takes all graphical parameters from screen points clicked with the mouse. A point of view is specified by clicking the screen positions of three corners of the data array. Zooming is accomplished by clicking opposite corners of the rectangle to be expanded. Prompts appear below the plot region to indicate what points and/or switches to click.

4) ViewData contains a call to DCBPress to allow generation of a one-page output file with a picture of your data. This can be annotated by Markup and printed by an appropriate server. With PressEdit, it can be editted into a report.

5) ViewData uses the new PlotStream package (to be released soon) to provide a display interface which is transparent to the average programmer; thus the program is easily modified to better suit your data viewing requirements.

6) ViewData is reasonably small, especially if one deletes unneeded routines from the various files which are loaded with it (MathUtil, SDialog, UtilStr, PlotStream, FractionProduct, DCBPress).

Getting and Running Viewdata:

Use FTP to retreive viewdata.run. If you need some sample data, use the FTP Load command to get Test.Data from ViewData.Dm (stored with sources). Execute ViewData and default all the parameters with CR to get a sample display. Using the mouse, follow the instructions of the prompts to zoom, redraw in a new orientation, or overview (zoom back out to the highest level). After you finish by pressing all three mouse buttons at once, you have the options of producing a press file, restarting (possibly with a new data file), or quitting.

Making a new Alto disk

This document describes procedures for creating a new disk, either by copying the disk or by using the File Transfer Program. It may be helpful to refer to documentation for Copydisk and FTP.

I.

The normal way to obtain a new, clean disk is to copy one of the Basic Alto Disks (Non-Programmer's, BCPL Programmer's, Mesa Programmer's, or Proofreader's) using Copydisk. You will need an Alto with a dual disk drive. Place the Basic Alto Disk into disk drive 0. Place the new disk into disk drive 1. Type

```
>NetExec
>CopyDisk
*Copy from: dp0
Copy to: dp1
```

Copydisk will copy disk 0 to disk 1, overwriting everything on the disk.

You can also copy the Basic disk from one Alto to another over the Ethernet. The CopyDisk documentation explains how to do this.

There should be a date on the label of the Basic Alto Disk which tells when it was last updated.

An alternative way of building a new disk from scratch is to erase it by means of the Install procedure, then use FTP to retrieve the subsystems and other files that you need.

First, bootstrap the NetExec by booting the Alto with the BS and single-quote keys depressed. Then type:
>Sys.boot

This will load a copy of the OS from the network. When it starts up, it will ask you if you want to install the OS; respond 'Y'.

Install will ask if you want the long dialog; respond 'Y'. Then it will ask if you want to erase a disk. Reply 'Y'. It will ask you for the name of the local file server and the name of the directory on that server from which to obtain files (the correct response to the latter question is usually 'Alto'). Finally, it will ask the usual questions about your name, the disk name, and the password.

When Install has finished initializing the disk it will run FTP to obtain the Executive. Now, to obtain current versions of the 'basic' software type
>ftp file-server ret/c <alto>newdisk.cm
>@newdisk.cm@

where 'file-server' is the name of your local file server.

After this has completed, to obtain additional software for a 'basic non-programmer's disk' type
>@npdisk.cm@

To obtain additional software for a 'basic BCPL programmer's disk' type
>@pdisk.cm@

To obtain additional software for a 'basic Mesa programmer's disk' type
>@mesadisk.cm@

II.

You can copy files from your old disk to the new one in two ways. One is to put them onto a file server

and retrieve them with FTP. If there are many, it is a good idea to package them into a dump file. The other way is to copy them from the old disk on one Alto to the new disk on another Alto. On your new disk, type
>ftp

On the Alto with the old disk, type
>ftp <Host name> store/c <filename1> <filename2> ...

<Host name> is the name of the Alto which has the new disk.

The easiest way to specify and transfer lots of files is to use DDS (if you have it on your old disk) to select the desired files, then issue the <Send to ...> command and type in the name of the Alto with your new disk.

Without DDS, a way to specify lots of files is to obtain a file with all your file names by typing
>*<control-X><control-U><return><return>

This will automatically invoke Bravo and read in 'line.cm'. You may then edit line.cm to exclude the files which you do not want to transfer and insert the necessary FTP commands, thereby creating a command file which may be invoked in the usual way. For example, at the beginning of the file insert
ftp <Host name> store/c

then delete everything except the files which you want to transfer. 'P'ut the command string onto a file. 'Q'uit out of BRAVO and type
>@foo@

where 'foo' is the name of the file which you just created with BRAVO. The selected files will be sent to the waiting Alto with the new disk.

Executing either variant of procedure I to erase and initialize your disk, followed by procedure II to transfer all of your files using FTP, is a good way to compact a fractured disk.

1. PARC Information

1.1. Getting Started

Each administrative group in Parc handles disk pack allocation differently. Ask your secretary how to get a disk.

A set of BASIC ALTO DISKS is kept in a rack near the Altos in the Maxc room. These disks are recreated once a week. The date when a disk was last created is on its label. Procedures for copying a Basic Alto Disk to your new disk are described in the "new disk" section of this document.

1.2. MAXC Directories for Alto Software

The <ALTODOCS> directory contains documentation for the subsystems and subroutine packages.

The <ALTO> directory contains current versions of all the Alto programs. Programs are normally kept in executable form; thus the CopyDisk program appears as <ALTO>CopyDisk.Run. In addition to the executable file, some programs also have a symbol file on <ALTO>. The symbol file for CopyDisk is <ALTO>CopyDisk.Syms. This file is useful to the author when something goes wrong with a subsystem, but it is not normally needed by users. Subsystems which need more than one file, either because they have overlays or because they need data files, should have the individual files stored, together with a command file which may be run to retrieve each file via FTP. The command file should have the extension .CM. Definition files have the extension .D. These files are useful only to programmers.

Subroutine packages are kept on <ALTO> with an extension of .BR or as "dump" files (extension .DM) if several files belong together as a package.

The <ALTOSOURCE> directory contains the source files for the subsystems and subroutine packages. It also contains the PUB files for the documentation which is on <ALTODOCS>.

1.3. Alto Software Maintenance Procedure

The maintainer of a subsystem or subroutine package handles a new or revised release in the following manner:

A. Copy a dump file with a name of the form SubsystemName.DM and the following contents to <ALTOSOURCE>:

1) The source files from which the subsystem may be created.

2) The command files which are needed to create the subsystem from the enclosed source, unless the creation procedure is "obvious." The following are the usual ingredients:

a) A command file containing statements to compile the enclosed source. Compiler messages should be written to a file. For example:

BCPL/F FOO.BCPL.

The filename should be in the format, COMPILEsubsysName.CM.

b) A command file to load the files which were produced in step a. For example:

BLDR FOO

The filename should be in the format, LOADsubsysName.CM.

If the subsystem is small, the two command files may be combined into one. The name should be in the format, CREATEsubsysName.CM. The following example will create the package for subsystem FOO.

BCPL/F FOO.BCPL; BLDR FOO

c) A command file containing statements to save all relevant files in subsysName.DM, e.g. the file DUMPFOO.CM would contain;

DUMP FOO.DM FOO.BCPL CREATEFOO.CM DUMPFOO.CM

B. When you have a change to make to documentation, or wish to introduce new documentation into the system, the following three steps are required:

1. Retrieve the relevant .PUB file from <ALTOSOURCE>. The file name is in the format, sys.PUB, where 'sys' is the name of the subsystem or subroutine package. If you are creating brand new documentation, start with the file <ALTOSOURCE>ALTODOCTEMPLATE.PUB, which contains the necessary Pub incantations and some instructions to authors.

2. Edit the pub file. Pass it to PUB-- a .TTY version of the documentation will be produced.

3. When you are finished, copy the pub file back to <ALTOSOURCE>, and copy the .TTY version to <ALTODOCS>.

Please be sure to copy the pub files from <ALTOSOURCE> afresh each time you edit them, because they may have been edited to produce expurgated versions (for distribution outside PARC), to produce indexes, remedy formatting problems, etc.

Please try to avoid needless references to PARC or Maxc facilities. For example, it is frowned upon to mention the <ALTO> directory as a place to find something. That is assumed for PARC users. Similarly, avoid needless references to GEARS or EARS.

C. Copy files needed for the new release to <ALTO>.

D. Send a message to Alto users describing the changes which will be effective with this release. The list of Alto users is on the file, <Secretary>AltoUsers.DL. The subject of the message should be the name of the subsystem or subroutine package. Try to keep the message short.

Passwords: The password to all Alto-related directories on MAXC is ISFWGI. Software maintainers are cautioned to alter only files for which they will take responsibility. Feel free to archive old versions, but please leave the current version of all files.

1.4. Alto Documentation

Formal documentation is provided in two forms: a "perusal" form, which can be conveniently typed at a T1 or VTS terminal on Maxc or perused with Bravo on an Alto, and a "notebook" form, which can only be printed on Ears or a Press printer, and may have fancy illustrations or fonts in it. Informal "message" documentation can be extracted from the <ALTO>MESSAGE.TXT file.

A. The "perusal" documentation is always stored on <ALTODOCS> under a file name like sys.TTY, where "sys" is the name of the subystem or package you are interested in. For example, the documentation for a subroutine package, FOO, would be found on <ALTODOCS>FOO.TTY. There is one exception to this rule: for very simple subsystems the documentation is in <ALTODOCS>SMALLSUBSYSTEMS.TTY.

B. The "notebook" documentation is packaged in larger packages to reduce storage overhead and to provide more manageable sets of documentation for printing. Currently, the following files are maintained in notebook-style:

Alto User's Handbook. This document is available only as a printed, bound manual. It contains the Non-Programmer's Guide to the Alto, and manuals for Bravo, Markup, Draw, and FTP.

BRAVO.EARS, MARKUP.EARS, DRAW.EARS, NSIL.EARS, GYPSY.EARS. Currently, these subsystems have their own separate Ears documentation.

OS.EARS. Operating System manual.

BCPL.EARS. BCPL manual.

SUBSYSTEMS.EARS, .PRESS. Documentation for most Alto subsystems. These are arranged alphabetically, with headings to indicate which system is being described. A directory at the front of the file contains documentation about very simple subsystems. The last section of this manual contains special information relating to Altos at PARC--where to find the software, how to maintain it, etc.

PACKAGES.EARS, .PRESS. This contains documentation for the software packages available for the Alto. A directory at the front of the file contains documentation about very simple packages.

ALTOHARDWARE.EARS, .PRESS. This is the "hardware" manual for the Alto.

TRIDENT.EARS. Documentation for the Trident disk controller.

These files are formatted, and should therefore be printed with

@EARS FOO.EARS -- or -- PRESS FOO.PRESS

C. The file <ALTO>MESSAGE.TXT contains all of the information which has been sent to Alto users with SNDMSG. Information about recent changes to a specific subsystem may be selected by using the 'subject string' option of the MSG subsystem. For example, you may type

MSG <ALTO>MESSAGE.TXT T S FOO

Or you can read the entire file by saying

File: <ALTO>MESSAGE.TXT

to READMAIL. Every six months this file will be purged and its old contents left on the next version of OLDMESSAGE.TXT.

1.5. Command Files

In addition to the subsystems, packages, and definition files, the following command files may be found on the <ALTO> directory:

NEWDISK.CM: creates a minimal system on a new disk. See the NewDisk procedure, in the Alto Subsystems manual.

DISTDISK.CM: creates the disk for distribution to other Xerox sites. NEWDISK.CM must be run first.

MESADISK.CM: creates a Basic Mesa Disk. NEWDISK.CM must be run first.

NPDISK.CM: creates a Non Programmer's Disk. NEWDISK.CM must be run first.

PDISK.CM: creates a Programmer's Disk. NEWDISK.CM must be run first.

PROOFDISK.CM: creates a ProofReader's Disk. NEWDISK.CM must be run first.